IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 19 December 2023, accepted 12 February 2024, date of publication 14 February 2024, date of current version 26 February 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3366455

RESEARCH ARTICLE

Enhancing Analytical Select Statements Using Reference Aliases

MICHAL KVET[1], (Member, IEEE), AND JOZEF PAPAN[2]
[1]Department of Informatics, University of Žilina, 010 26 Žilina, Slovakia
[2]Department of Information Networks, University of Žilina, 010 26 Žilina, Slovakia

Corresponding author: Michal Kvet (Michal.Kvet@uniza.sk)

This work was supported
in part by the Erasmus+ Project under Project 2022-1-SK01-KA220-HED-000089149, in part by the Project title: Including EVERyone in GREEN Data Analysis (EVERGREEN) funded by the European Union, and in part 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.

ABSTRACT Data analytics is an inseparable part of the current information systems. Various tools can provide the analysis and produce results in any graphical form, enclosed by the complex filtering. Behind the scenes is a data layer and methods for accessing, manipulating, and processing data. SQL language and databases can serve that. This paper deals with data processing and performance optimization by focusing on function processing and reference. It points to the existing syntax and statement execution steps but provides various enhancements and performance optimization. Existing feature management solutions include result caching, function-based indexes, virtual columns, materialized views, or optimization of the functions to be directly applicable in SQL or PL/SQL limiting context switches. Oracle Database 23c introduced various performance enhancements and a new approach to column and expression aliases. Our proposed solution focuses on identifying and extracting aliases, storing the references in the memory and database layer, optimizing the transfer between them by swapping, as well as checkpointing and function call migrations. It provides a robust layer and complex architecture enclosing the management by the transactions. Each layer is critically discussed by pointing to the performance, structural advantages, and limitations. Complexly, our proposed architecture brings significant performance benefits for complex analytical queries but can also be applied in online transaction processing.

INDEX TERMS Analytical databases, column aliases, function references, indexing, performance.

I. INTRODUCTION

Data analysis is an important part of information technology. It refers to the process of getting raw data and converting it into information [1], which is then used for reporting, pattern detection [2], decision-making [3], or making prognoses [4]. Currently, data is collected hugely, and structures and relationships are becoming more and more complex [5]. There are many tools for supervising the data analysis process, like PowerBI, Oracle Analytics, R, etc. The main advantage of these tools is pre-prepared wizards, so anybody can perform analysis, even those who are not professionals in this field. These tools can be deployed on-premises or in the cloud. Currently, developers and data managers are more oriented towards cloud technologies [6], which move the environment and technical background management to the cloud vendor. Thus, there is no need to worry about the hardware, patching, and updating, but mostly the solution's scalability. Pay-as-go principles further emphasize this because users pay only for the actual use of the resources, dynamically as needed [7].

Thus, complex data analysis can be performed. The main advantage of the mentioned tools is the ability to process and provide any result based on the input data. Users do not need to focus on the data layer, just the provided outputs. The focus is on the results, not the process of obtaining them, calculations, and data access. This can, however, have many disadvantages in terms of performance [8]. Namely, there are two crucial aspects – data layer and Select statement

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Asif Naeem.

execution, generated behind the scenes [9], [10]. SQL is a powerful language for manipulating data and providing complex evaluations, querying, and reporting, operated by the Select statements. This paper focuses on the performance of the data analysis in the Oracle Database environment by pointing to the column, expression, and function aliases and their applicability through the execution to make the process easier, but mostly more powerful by reducing the costs and processing time demands. The aim is to extract the aliases and make them usable across the whole process by enhancing pre-fetched function results.

This paper is structured as follows. Section II deals with the background of the data analysis and environment description. In section III, performance and data reference problems are stated with the reflection on the order of steps and execution plan of the statement. Section IV deals with the existing solutions and related techniques. Section V describes proposed solutions, followed by the performance evaluation study depicted in section VI.

II. BACKGROUND

Relational databases were introduced at the beginning of the sixties of the twentieth century. Throughout history, massive development, tuning, improvements, and extensions have been undergone. They were initially used for conventional data management by emphasizing data structures and normalization. The relational paradigm is based on the entities interconnected by the relationships. Applying this concept properly, including data integrity, consistency, and overall performance, is crucial. Limiting duplicates and anomalies brings reliability, data layer optimization, storage properties, and demands. Therefore, data models are encapsulated by the normalization process and covered by transaction support. Database transactions are important for data management, taking ACID properties [11] – atomicity (A) ensuring the transaction is executed as a single inseparable unit, so either the whole or nothing is applied. The consistency (C) aspect checks all the constraints are applied. Thus, it is impossible to approve the transaction and make it durable if any constraint is violated. Isolation (I) ensures that only approved transaction data are spread and visible across the ecosystem. Running transaction data is secured and visible by that transaction only. The durability (D) aspect guarantees that the effect of the approved (committed) transaction is permanent, even after the system failure. Thus, transaction definition is a milestone of database technology, delimited by the transaction logs, which describe the whole data management process using change vectors. Transaction data processing is an operational unit that treats and evaluates data from the external system through the database input layer. Data are considered, filtered, and stored in the transaction-oriented database as operation data. In the past, they were mostly related to the conventional layer. Thus, only current valid states were treated. In the eighties of the 20th century, the first primitive temporal paradigm was introduced [12], based on the object-identified extension by the validity time frames. Although the evolution and timeline reference could be done, practical implementation and storage efficiency was limited because each change operation forced the system to release a new object state, covering all attributes. And even in those situations, when only one attribute or generally a subset of the tables attributes are changed. Consequently, many duplicate tuples were stored.

An autonomous transaction database, which is part of cloud technologies, was introduced by Oracle Corporation. Thanks to autonomous processing, database administration, patching, upgrading, securing, and continuous availability, it is shifted to the cloud vendor's responsibility. There are two other database types among transaction databases: JSON–oriented and data warehouses. Our proposed solution is mostly analytically oriented (Fig. 1). However, it can be fully deployed on any database. In this paper, we focus on the Oracle Database for several already introduced reasons. From the architecture point of view, the most significant reason relates to the complexity and enhancements of this database type so that these features can be used as a reference for performance evaluation and comparison. The most powerful database system – Oracle database 23c – is used for the performance evaluation reference to declare the performance of the entire solution and its comparison in real application practice.

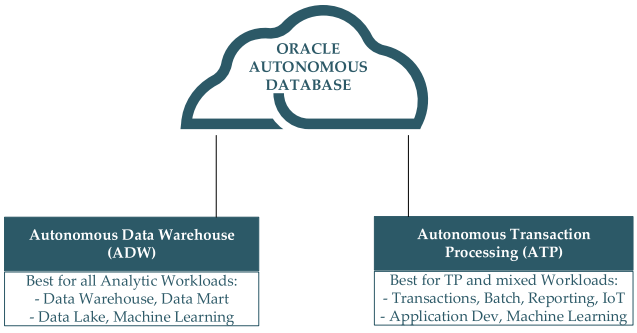


FIGURE 1. Oracle autonomous database types.

Oracle's autonomous data warehouse is the worlds first and only autonomous database directly optimized for analytic workloads, starting with data warehouses [13], including data marts, lakes, and data lakehouses. It is an ideal solution for any user type – business analysts, database specialists, managers, and nonexperts can rapidly discover business insights and manage scalable systems to analyze the data. It runs on Exadata architecture, which lowers the costs by an average of 63% [14], [15].

The background of the data analytics relates to data access and evaluation. SQL language is a standard for database creation, operation, and manipulation. Its the most often used approach for data management in this era. The first commercial relational database system using SQL was released in 1979 by Relational Software, Inc. (currently branded by Oracle). The SQL language soon became an inseparable part of most relational database systems. In 1986,

the ANSI institution standardized the SQL language under the designation SQL-86. A year later, in more or less unchanged form, this standard was also ratified by the ISO organization (SQL-87). In 1989 and 1992, revisions were designated SQL-89 and SQL-92 (also SQL-2). Significant changes came in 1999, when the SQL:1999 (SQL-3) standard included recursive queries, triggers, regular expressions, non-scalar data types, object-oriented properties, etc. The SQL:2003 [16] version mainly supported XML, standardized sequences, columns with automatically generated values, and so-called window functions. Among the standardization, many enhancements and extensions are applicable for the specific system through its local dialect. One of them is associated with the column, expression, and function call aliases stated in the Select clause of the statement. Applying SQL standards makes it impossible to refer to the aliases in the same statement, in clauses Where, Group by, and Having, because these clauses are evaluated for extracting expressions in the Select clause. Hence, they are unknown to the database processor in those phases. However, Oracle uses another approach and enables alias references anywhere in the statement in Oracle 23c, released in April 2023 [17], [18], [19]. This approach is used as a reference for our proposed solution, declaring the performance and limitations. The next section deals with the Select statement definition, followed by its execution, index references, execution plans, and function processing.

A. PROBLEM DEFINITION RELATED TO THE EXECUTION PLAN

The data analysis aims to get data insights, get the data, analyze them, and provide results. It is done by obtaining the data from the database, evaluating them, and building outputs as a result set. In SQL, it is done by the Select statement. The next block shows its syntax and individual clauses [10], [20]:

- *Select* clause deals with the list of columns (attributes), expressions, and function calls, representing the data provided in the result set. By default, the names of the attributes are copied from the definition, so the output headers do not need to conform to the expectations. Therefore, the aliases specified just after the definition can enhance each attribute, expression, and function call, forming the header for the result set or consecutive processing. In addition to the formal side, an alias also has other properties. First, it hides the definition and method of calculation and processing of the given expression, as well as the possibility of easier reference during further processing. Lastly, an alias may be necessary due to association with a view, table or materialized view.
- *From* clause specifies the source tables or views, optionally defined by the join operations delimited by the interconnection conditions. The ANSI joins are recommended to make the joining process clear. The join order can either be left to the database optimizer to select the most appropriate join plan based on statistics. The second solution is to apply the order from left to right as specified in the query. The approach is based on the system selection or query hint can be used to navigate the processing.
- *Where* clause is used to hold conditions filtering the records, it extracts only rows that fulfill all conditions. The logical sum or count interconnects multiple conditions – AND or OR.
- *Group by* clause is primarily associated with the aggregate functions calculated for each defined group. This clause takes all rows with the same value of the set into one group. In the result set, each group is represented once. Therefore, the number of rows is reduced to the number of groups.
- *Having by* clause is a specific filter variant executed at the end of the processing. Thus, it is used for conditions based on aggregate functions. Although generally, any condition can be placed there, it is not recommended because of postponing filtering and requesting to process larger data sets than necessary.
- *Order by* clause takes the obtained result set and sorts it based on the criteria. Without that, data are not sorted and produced in the order in which they were accessed in processing.
- *Limit* clause reduces the the size by getting only a portion of data instead of the full result set, which fulfills all the conditions stated in the Where and Having clause. Thus, the number of columns remains, but the result set's cardinality is reduced.

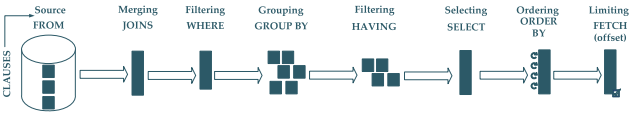


FIGURE 2. Select statement procesing steps.

When the statement is defined, it is necessary to execute it by locating the data, joining, filtering, sorting, and producing the result set. Select and From clauses are mandatory; the rest are optional. The data retrieval execution requires individual clauses to be handled. Fig. 2 shows the order of evaluation of individual clauses. It starts with the source tables, which are merged and filtered. Individual tables can be interconnected using the ANSI join specified in the From clause, which is preferred due to the readability and reliability to ensure the connection's correctness and completeness. But in principle, the From clause can state only a list of tables, and the connection details are then specified in the Where clause as a conventional condition. One way or another, the goal is always to reduce the amount of data for further processing as much as possible. Therefore, if not stated explicitly, the order is based on the estimated cardinality. There are two hints – *Ordered* (the order of table joining is strictly defined in the Form clause) or *Star* – the order is based on table cardinality (the largest table is joined at the end) [21], [22], [23].

Group by definition is optional, defining the Group for which the aggregate function is calculated. Aggregate function can also be used for the filter. After applying groups and calculating aggregate functions, a Having clause is applied, considering the additional filter. After all previous phases (joining, filtering, grouping), the Select clause is processed by identifying the structure of the result set. In this clause, aliases can be defined. Typically, users want formatted data, like date element extraction, concatenating string, rounding, etc. Such data, however, can be used as filters, consecutively placed to the Group by clause. It requires copying the definition since the alias stated in the Select clause cannot be generally used in the Where, Group by, or Having clauses because those clauses are evaluated before treating the Select clause. This limitation can lead to many errors and bring additional coding demands.

Furthermore, if an expression or function call is stated multiple times in the query, it is calculated several times, even with the same parameters. Thus, it brings additional processing time demands and costs, mainly caused by the content switches between SQL and PL/SQL function calls. Oracle 23c brought the ability to reference aliases anywhere in the Select statement, discussed in section IV.

The execution plan represents individual operations performed by the SQL query processor. Whereas SQL language is declarative, many alternative ways of executing a given query with widely varying performance usually exist. Database optimizer prepares various access paths, from which the heuristics select the best option. The input data for the decision-making are descriptional, covered by the database statistics, comparing the estimated and sequential block processing costs. This execution plan is then temporarily stored in the Shared pool of the instance memory, so repeated use of a given query can use a pre-calculated plan.

One of the key decisions of the database optimizer highlighting the execution plan relates to the index usage [24], [25]. Index is an optional database object associated with the table to speed up the search process for a specific record. By default, B+trees are used, but bitmap indexes are also hugely used in the data warehouse environment because the bitmap operations are fast so data access can be strongly beneficial. On the other hand, any change in the data usually requires complete bitmap index reconstruction. Indexes can refer to the table attributes from the expression and function call reference, but expressions and functions can also be indexed. The data retrieval process can benefit if the functions are treated in the index by reflecting the parameters usage – pair – function and results.

Fig. 3 shows the B+tree index structure, consisting of one root node, internal nodes, which hold the index key values and pointers to consecutive layers, and a leaf layer, which consists of the addresses of the rows (ROWID) for each key value. Table Access by Index Rowid method is the fastest way to locate and obtain a row from the database because it directly navigates to the data file, block, and row position inside the block. ROWID takes 10 bytes [10]. Although it is

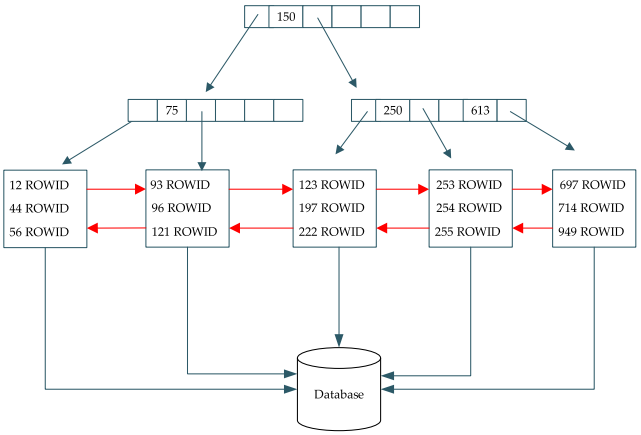


FIGURE 3. B+tree index structure.

a precise address of the row, the referenced data block often does not hold the data of interest, which is caused by the data migration. Namely, after the change operation, the original row size can be extended and does not need to be able to be placed in the referenced block. Therefore, the original block takes a pointer to another repository – the block in which the row resides. Generally, the path to locate the row can contain multiple blocks, and overall performance is degraded. To limit such a situation, it is necessary to reconstruct the indexes regularly to eliminate data migrations.

Indexes are also necessary for table joining to ensure smooth data merging among the regular data access. Typically, tables are joined together based on the primary and foreign keys, but generally, any set of attributes, even expressions, can be used for serving the join. There are three methods: operating the overall workload and estimated costs. The most suitable option is a Merge join, which benefits from the join key indexes enhancing both tables, so the input data are pre-sorted. The Nested loop is based on one table-taking index; the second must be fully scanned. If that table is small, additional performance impacts are not huge compared to the task of dynamic index creation.

On the other hand, for large tables, it is beneficial to divide them into smaller portions – buckets, done by the hash function getting the bucket assignment. The hash function is called before the statement processing, so instead of scanning the whole table, only one bucket needs to be sequentially scanned. Besides, the database optimizer selects the most appropriate hash function, so the bucket is not too large for the scanning.

Moreover, it can be done in parallel [10], limiting the additional demands. On the other hand, it is always necessary to build buckets and make data assignments. The last method, available in the Oracle Database, is an adaptive join, which combines techniques of the Nested loop and Hash Match. Changing the access method dynamically is possible based on the estimated and real costs. It is used when the computed statistics are not correct and up to date - the amount and structure of the data are significantly different compared to the stored assumptions.

SQL language is a non-procedural language, so the database optimizer must select the most suitable option to locate, access, and merge data sets to provide outputs reliably, in the proper format, and at a suitable time. Therefore, indexes were developed to simplify the scanning process by limiting the necessity of sequential scanning and memory loading. As evident, individual data access paths and join methods are treated in the initial phases. In this phase, individual functions can be called defining the filtering. Thus, any column aliases should be identified before to enable the benefits of using them among the whole statement. It could, however, influence the execution plan and data methods. The next section summarizes existing solutions by highlighting function management, content switches related to the PL/SQL code processing, and techniques to observe function results.

III. EXISTING SOLUTIONS AND RELATED TECHNIQUES

Even with the advent of databases and the SQL language, many techniques and attempts existed to optimize queries and speed up processing. The relevant techniques were usually expanded and generalized, becoming part of the SQL standard, which continuously evolved over the decades. On the other hand, each database system creates its dialect and comes up with different techniques, enhancements, and extensions. The Oracle database will be used for evaluation and reference purposes since it applies all relevant techniques to optimize the performance in dynamic systems. Furthermore, Oracle Database 23c brought various performance-enhancing techniques focusing on the SQL language. In this section, the relevant techniques and existing approaches related to function processing are referenced. They will also be used in the performance computational study to focus on the usability and applicability of the proposed solutions and point to their limitations. This section is divided into multiple streams.

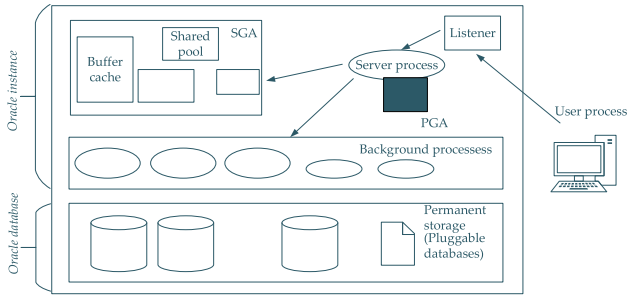


FIGURE 4. Database system architecture overview.

A. RESULT CACHE

The architecture of each database system consists of the database - data layer holding the physical data and management system called instance (Fig. 4). Oracle database is stored in the permanent storage, either in the files system operated by the operating system or in raw data sources (ASM segment), which is self-managed by Oracle [26]. The database instance is defined by the memory structures and background processes operating the memory and database themselves. Inside the Shared Global Area of the database instance, the Buffer cache and Shared pool are relevant for the function calls inside the statements. Buffer cache is a work area for executing SQL statements. The block itself is the finest processed granularity inside the relational databases, which must be transferred from the database into the instance memory for the evaluation and building result set. The system must locate a particular data block (done sequentially or using an index), which is then placed in the buffer for processing by I/O loading. If the function call enhances the processing, its parsed definition needs to be obtained first. To prevent repeated reading, the data dictionary cache of the Shared pool memory structure serves the metadata of the recently used object definitions. Besides, Oracle Database 11g, introduced in 2007, brought a new feature to store the results of the functions in the Result cache part of the Shared pool. It limits the necessity to execute the same query multiple times. Each query is identified by the statement identifier (Plan hash value) for the consecutive reference. This structure holds the whole statement. There are three modes – auto, manual, and force [27]:

```
alter system set result_cache_mode = [auto | manual | force ]
```

The default option is manual, by which the query hint can enhance the statement to navigate the system to store the execution plan in the Result cache:

```
select /*+RESULT_CACHE*/ list_of_attributes,
                         list_of_expressions, function_calls
  from ...
```

Part of the Result cache points to the function calls and stores pairs – functions with the defined parameters and their results. Admittedly, those functions must be deterministic, so for the input parameters, the same result is always obtained, which is declared by using the Deterministic keyword specified in the function header.

Oracle ensures an intelligent aspect of the Shared pool, so if the definition of any object is changed, referenced values are automatically invalidated.

The data loading into the instance memory Buffer cache is supervised by the Server process associated with the client session. For each query, the database optimizer calculates the execution plan. The query definition itself is hashed to obtain the Plan hash value, which is, among the result cache reference, also used for identifying parsed queries stored in the Library cache of the Shared pool. Similarly, if the definition, structure, or index set is changed, precalculated plans are invalidated. Furthermore, existing execution plans are continuously reevaluated to ensure the best suitable execution process [23], [26].

Fig. 5 shows the query execution processing steps related to the instance monitoring and caching. Concluding, enabling result caching reduces the amount of repeated execution of the same code, as well as multiple calls of functions with the same parameters. In this case, the aliases specified in the Select clause are not directly applicable in the

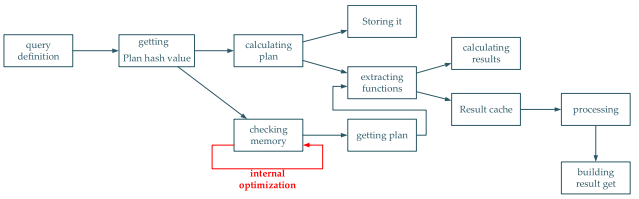


FIGURE 5. Query execution steps.

Where condition, and thus, a repeated definition is required. However, at the execution level, the function will not be called repeatedly, but the already precalculated and stored value in the instance's memory space will be used. Naturally, it is necessary to ensure sufficient capacity of this structure to cover different parameter variants. On the other hand, a smaller capacity brings additional costs for locating and referencing record results. The limitation of this approach is based on the definition itself. Since the hashes are calculated, it is important to specify the function in the same manner. Namely, any change in the format causes a change in the hash value, so the pre-stored value would not be mapped, forcing the system to calculate the result on the fly.

B. FETCHING FUNCTION RESULTS USING VIRTUAL COLUMNS

In Oracle Database 11g Release 1, virtual columns were introduced by associating expressions with the table definition. Thus, when querying, virtual columns look like ordinary stored columns, but their values are derived rather than physically stored in the database. Although the virtual column values are calculated on the fly, it mainly benefits because the values from the virtual columns are stored in the Result cache like ordinary columns; its definition is directly stored in the table definition and obtainable through the system table metadata. So, there is no additional need to reference function results and calculate function call hashes. Furthermore, suppose the virtual column is not based on unique values. In that case, the calculation of the virtual columns can be fetched in the private area for the query by using the mapping table – virtual column input and result value. From the performance point of view, it provides better results since the data are grouped within one structure. The comparison of the architecture of the virtual column management and function call caching is depicted in Fig. 6.

In this solution [10], [28], [29], it is still impossible to refer to the aliases in other clauses of the Select statement. However, virtual columns act very similarly to the function call aliases. At the same time, the definition is stated only once, although located externally from the data themselves, placed in the table description metadata. The syntax is stated in the following code block:

```
column_name [datatype]
   [generated always] as (expression) [virtual]
```

In conclusion, the virtual column's main advantage is storing the pure table data and function call calculations through the virtual columns together, inline in the query Result cache,

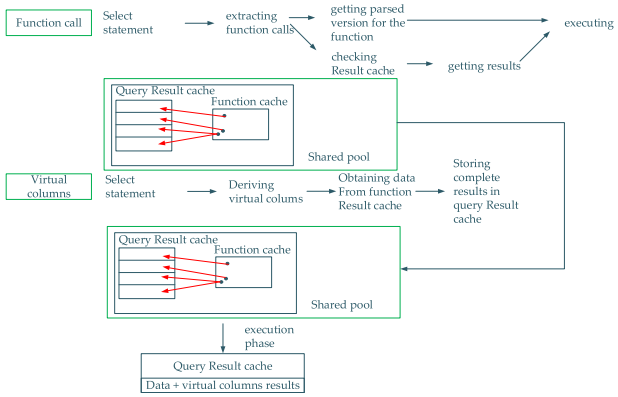


FIGURE 6. Function call vs. virtual column management.

compared to the function call management, in which two separate structures must be referred to in the Shared pool. Individual rows are interconnected to the function results via connector pointers. This solution's limitation relies on the function Result cache capacity and the eventual necessity of swapping records to disk. Swapping may also be necessary in the case of a virtual column, but the entire record in one structure is swapped with no other references.

C. MATERIALIZED VIEWS

Materialized view (snapshot) is a table segment in which content is periodically refreshed [10], [30], [31], [32], [33]. Similarly to the table, it physically stores the data originating from data table sources, enhanced by the function calls and results, directly stored within the data. Thus, there is no need to call the functions, since the results are directly present within the data set. Materialized views can refer to the query as well as functions. Tables can be local or remote; tables using database links can also be applied. Although primarily intended for data replication in the data warehouse environment, it brings sufficient power for the complex, function-based enhanced queries in transactional and analytic environments. The keyword defines it as Materialized in the view header. The refresh can be done for each transaction end–commit or on demand. There are three refresh types – fast (a fast incremental refresh is attempted. It requires a materialized view log to be present for the referenced tables. If not defined, the creation of the materialized view fails). Complete refresh asks for truncating the existing view and creating a new one. The force option is the default. A fast refresh is done if possible (materialized view log is defined). If not, a complete refresh is done. The syntax is following:

```
CREATE MATERIALIZED VIEW view_name
  BUILD [IMMEDIATE | DEFERRED]
    REFRESH [FAST | COMPLETE | FORCE]
      ON [COMMIT | DEMAND]
        [[ENABLE | DISABLE] QUERY REWRITE]
    AS select ...;
```

A materialized view log is a schema object that records changes to the master table data by allowing the refresh of the materialized view immediately. It works similarly to the

transaction log, but it points only to the particular tables. It is created on the table level, and addressing is expressed by the primary keys. The architecture of the materialized view management is in Fig. 7.

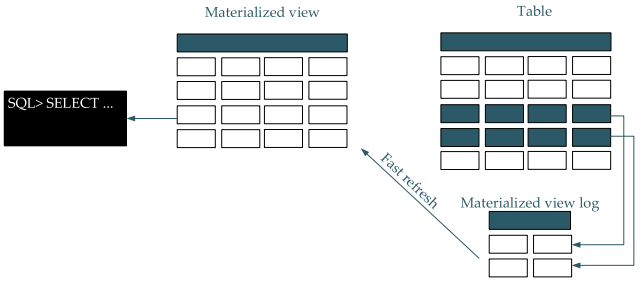


FIGURE 7. Materialized view and log architecture enabling fast refresh [34].

Materialized views enhanced by the logs bring sufficient power for dealing with complex calculations in an analytic-oriented environment. To make materialization reliable, data must be refreshed to ensure actuality, not only read-only historical snapshots describing the static data portion. Thanks to the definition, calculated data are directly part of the structure, so there is no need to point to another repository, nor is any calculation and context switch necessary. Thus, data management is fast and reliable if the materialized view logs are defined for the source table, giving the ability to refresh data in a fast mode. Otherwise, there can be huge costs for the complete refresh, mostly in the dynamic transaction-oriented system, where many changes occur. The main limitation of this approach relates to the additional storage capacity demands, whereas the data are physically stored in the database. Namely, data are covered by the core source tables and indexes and shaped in the materialized views. Managing the consistency of these states is an integral part. However, the database system itself takes care of that.

D. REFERENCING FUNCTIONS IN SPECIFIC DATABASE REPOSITORY–GROUPING BASED ON STATEMENT ID

As stated, database systems allocate the space for storing function results in the instance memory. Thus, if a particular reference is already present in the Result cache structure of the Shared pool, there is no necessity to reload that function and calculate it multiple times. Instead, the stored result is directly taken, reflecting the prerequisite that the function is deterministic [35]. The problem with this solution is associated with the memory requirements. To serve the requests, the huge capacity of the instance memory must be considered. Namely, if the Result cache is full, available memory capacity can be dynamically reallocated, and the Shared pool can be extended. However, memory capacity is limited, so that it will be filled sooner or later due to the number of data, extended requirements, and more complex analytics. There may be better solutions than expanding and increasing the available memory. Namely, in the on-premise world, it is impossible to expand the memory indefinitely; the hardware configuration, available sources, and limitations determine it. Deployment in the cloud environment is more straightforward. However, CPU and memory are the most cost-demanding, compared to the storage. Therefore, additional techniques were developed to serve as the repository for the function calls. The advantage of this approach is that it is based on storing data in the database instead of memory; thus, after the instance of failure, such data are still present and accessible. Moreover, storage capacity is easier to extend; disc prices are low for the on-premises cloud capacities, and offers are widespread and cheap.

The architecture of that solution is formed as a multi-index organized table. There are two levels of keys. The first key acts as a partition definer, delimited by the function identifier and marked by the method ID provided by the database system during the compilation. The second key forms the B+tree. It consists of the parameters for the particular function, enhanced by individual values. The leaf layer takes the return value of that function. Thus, for each function (partition), a separate local index is created because of the parameters, which generally differ across the functions. Similarly, the output values can have various formats.

Additionally, there is a mapping function object considering the function hash values. Namely, after the transfer into another system, method identifiers are changed. To make the module relevant and usable after import, only the mapping module is recalculated, and the core structure remains the same.

The defined solution is robust and can easily cover any function call. Reflecting the architecture and B+tree orientation, a particular result segment can be easily identified. Any function call not indexed automatically extends the structure to ensure consecutive calls can take the already calculated value without executing the function multiple times with the same parameters. It would, however, require index rebalancing, while the B+tree is always balanced [35]. That would influence the performance of the query. Therefore, to limit additional costs, applying new function calls and resulting values to the index are done in a separate autonomous transaction and do not synchronize with the main transaction. After applying a new value to the index, the transaction ends by reaching commit automatically.

The described architecture allows you to manage any function call by invoking each function with the defined parameter set only once. However, what about retention? With that approach, disc storage demands would continuously rise; even functions invoked only once would be permanently stored. Therefore, several techniques were proposed to limit disc storage demands [36]:

- *Time retention* – the function remains in the storage reference module for a minimum of seconds. The definition remains if there is still free storage capacity; otherwise, it is freed. The time reference point can be either the timestamp of the node creation (first call of the method) or it can refer to the last invoke.

- *Call retention* – it defines a minimum number of other function calls to allow freeing its definition. Each time the function is called, the counter is restarted.
- *History retention* evaluates the number of times the function with the defined parameters was invoked, correlated with the total number of invokes. Thus, if the function is hugely used, it is more preferred. On the other hand, it also considers the time flow. The use of the function can change over time.
- *Baseline* – storing function in a baseline ensures unlimited accessibility through the stored object and will never be freed.

These stated methods define the priority queues influencing the data portions to be flushed, due to the capacity of the structure.

The selection can be done on the system, schema, or function level:

```
alter {system | schema | function function_name}
  set retention {{creation | invoke}
   time value seconds | call value |
   history value {seconds | calls} | baseline}
      max storage value {unlimited | value {MB | GB | TB} };
```

If the maximal capacity is reached, no more functions results are collected before freeing it.

Although it is impossible to refer to the expression alias in the same statement, the described solution limits invoking a particular function multiple times. Any retention policy ensures that the stored object frame covers functions used in one statement.

E. LIMITING CONTEXT SWITCHES

SQL statement can consist of the PL/SQL code called a function. SQL code calling a function must pass the function call to the PL/SQL engine for the execution. The results are fetched and passed back to the SQL calling environment. Hence, two context switches are present for one function call. Generally, multiple functions can be executed, optionally enhanced by various parameters. This is a consequence of calling procedural language compiling source code into byte code to be interpreted from the non-procedural SQL environment. In the past, there were multiple embedded SQL parsers for different uses and environments. Starting from Oracle 8i, a standardized common SQL parser accepting any workload is available. Although it is always recommended to use native SQL functions whenever possible, there are plenty of circumstances where your code must be invoked.

Pragma UDF (User Defined Function) navigates the compiler to consider the parsing of PL/SQL function primarily used in SQL statements. Based on [37], it can bring performance improvement, reducing costs and context switches. The definition is part of the function declaration section. The syntax is stated in Fig. 8.

Pragma UDF optimization feature was introduced in Oracle 12c in 2012. Based on the performance evaluation

FIGURE 8. PRAGMA UDF syntax.

study defined in [38], PL/SQL Pragma UDF function can reduce the total demands by up to 76%, compared to the traditional PL/SQL. On the other hand, native SQL always brings the best suitable solution. For the data type conversion function, embedded Oracle functions reduce the processing demands by 63% on average (compared to the function definition with the Pragma UDF definition). Considering traditional PL/SQL and native SQL, taking available existing functions lowers the processing time demands by up to 91% [38].

F. FUNCTION-BASED INDEXES

The function-based index does not cover direct column values, but it is created on PL/SQL functions and expressions, forming the lookups on columns referenced by PL/SQL functions [39], [40], [41]. It can invoke any function which is recognizable in SQL. User-defined functions must be deterministic and stated explicitly in the function header. When a query that could benefit from that index is passed to the server, the query is rewritten to allow the index to be used. Indexing original column values leads to using the Table Access Full method, forcing the system to perform sequential data block scanning. The Index Range Scan method, followed by the Table Access by Index Rowid, uses a function-based index. The function-based index can consist of any number of columns or function calls and can be concatenated. Remember, function-based indexes require more effort to maintain than regular indexes, so having concatenated indexes in this manner may increase the incidence of index maintenance compared to a function-based index on a single column.

Advantages of function-based indexes:

- A function-based index speeds up the query by giving the optimizer more chance to perform an index range scan instead of a full one. Note that an index range scan has a fast response time when the WHERE clause returns fewer than 15% of the rows of a large table.
- A function-based index reduces computation for the database. If you have a query that consists of an expression and use this query many times, the database has to calculate the expression each time you execute the query. To avoid these computations, you can create a function-based index with the exact expression.
- A function-based index helps you perform more flexible sorts.

Limitations:

- The database has to compute the result of the index in every data modification which imposes a performance penalty for every write.

- The query optimizer can use a function-based index for cost-based optimization, not for rule-based optimization. Impact of implicit conversions is discussed in [42].

G. IDENTIFYING AND APPLYING COLUMN ALIASES

The Oracle Database significantly changed the alias management and handling concept in April 2023. Oracle 23c introduced an interlayer responsible for extracting column and expression aliases specified in the Select clause of the statement at the beginning of the execution [43]. Thanks to that, aliases are visible across the whole statement, making them referencable in any clause. The extended process of the statement execution is shown in Fig. 9. It is done by direct mapping and statement rewriting – aliases are extracted, and the definition is placed to any occurrence. It generally allows one to refer to the alias anywhere in the statement, even in the Select clause, making the function call nesting easier for the definition. However, it simplifies the user experience and definitions; before processing, aliases are replaced by the original definition, resulting in the same, already stated problem – one function is placed in the query multiple times. Referring to the Result cache can only partially solve the problem, while locating the return value for the function call with the defined parameter set is always necessary [44].

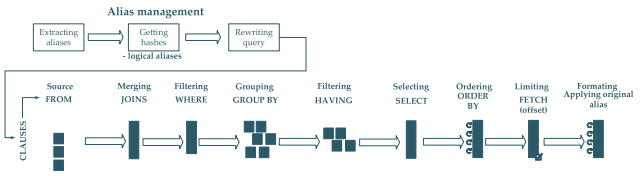


FIGURE 9. Extended select statement execution process.

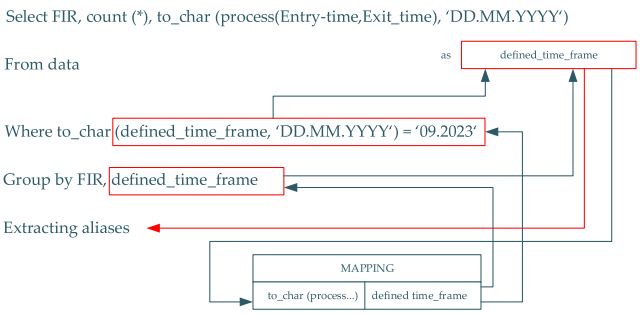


FIGURE 10. Select statement transformation example.

Fig. 10 shows an example of the statement transformation used for the flight monitoring system and flight information region (FIR) assignment to calculate flight efficiency and impacts on the environment, fuel consumption, gas, etc.

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.

H. MAKING ALIASES VISIBLE

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.

This 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. The architecture and process flow are expressed in Fig. 11.

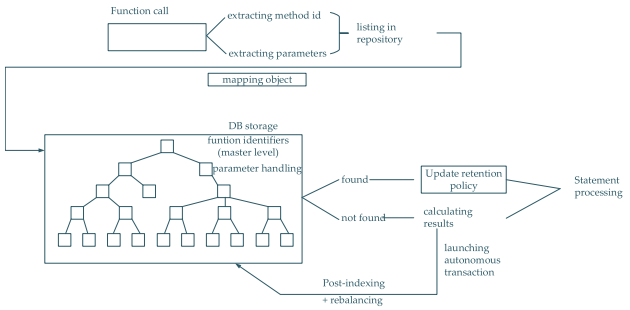


FIGURE 11. Architecture and process flow.

I. REFERENCING SELECT CLAUSE IN THE GROUP BY SECTION USING POSITIONAL NOTATION

The preceding parts mainly focused on data source management and filtering using Where clauses. The data aggregations and partitions define another optimization option by highlighting the group management and conditions based on the aggregate functions placed in the Having clause. In previous releases, it was necessary to repeat full expression and function call references in the Group by and Having sections. From Oracle 23c onward, it is possible to use column aliases and positional notations in

the Group by and Having clauses [45]. The same principles of identification compared to the solution stated in G are used, but the management is enhanced by the positional references, which require additional modules. To make positional references applicable, session-level parameter group_by_position_enabled must be set:

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

By tracing the final statement for the execution, positional references are replaced by the original calls:

```
alter session set events '10053 trace name context forever';
-- original statement
select initcap(d.dname) as department,
            count(*) as amount
   from  dept d join emp e using(deptno)
     group by 1
       having amount > 10;
-- transformed statement
SELECT INITCAP("D"."DNAME") "DEPARTMENT",
             COUNT(*) "AMOUNT"
   FROM   "KVET"."DEPT" "D",
                "KVET"."EMP" "E"
      WHERE  "D"."DEPTNO"="E"."DEPTNO"
         GROUP BY INITCAP("D"."DNAME")
            HAVING COUNT(*)>10
```

It is evident that fully qualified names are used in addition to rewriting references through positions and aliases. Furthermore, ANSI join is rewritten to the Cartesian product specified in the From clause. The joining condition is extracted and treated in the Where clause. This allows further optimization during the source table data access, extraction, filtering, and merging.

IV. PROPOSED SOLUTION

The proposed solution aims to optimize the performance of the alias management in synergy with the user experience. The solution cannot negatively affect the definition of the query. Existing queries must remain in the original structure even in the expanded environment - without changing the statement. Thanks to this, the solution can be deployed in any environment without intervening in the existing code, which would be practically impossible in a real environment - it would bring significantly increased costs and the need for integration tests and solution verification. Moreover, the proposed solution would not be applicable if we only had access to the final code compilation. Therefore, when implementing our solution, we focused on preserving the non-procedural nature of the statement definition as the basis of the SQL language.

The proposed solution is based on existing approaches, which are taken as a basis. Still, it introduces its own architecture and data structures to make aliases not only generally identifiable in the query but also applicable and identifiable in individual clauses, whether of a direct Select statement or in individual nested query branches. The aspects of query usage prioritization, generic local addressability, dynamic migration, command-based partitioning, and reflection within the private part of session memory (Private Global Area (PGA) are introduced. All these aspects are step-by-step defined in the following subsections.

The proposed solution architecture influences the whole processing therefore, we will introduce and discuss individual modules step-by-step:

A. IDENTIFYING AND EXTRACTING ALIASES

Usually, column, expression, and function call aliases were extracted and treated during the execution of the Select clause of the statement. It was primarily used to shape the column format in the result set. Later, however, queries became more and more complex and nested, formed by analytics and reporting, attempting to refer to the aliases in a wider form. Oracle 23c extracts the aliases in the first phase, so the rest of the query execution waits to finish the extraction. This can, however, bring various limitations, especially in cases where the extracted aliases are not later used and referenced. Our proposed solution introduces a new background process – Query Alias Extractor Master (QUAM) process, which is a supervisor for the whole activity and balances the workload activity across the set of the worker processes – Query Alias Extractor Worker (QUAWn). There is one master (control) process. There can generally be an unlimited number of workers, which can be, in addition, created and released dynamically based on the ratio of the queries, number of aliases used there, etc. Their aims are to detect and extract aliases from the defined query and create a list of them for the references, as stated in the B and C subsections of this section.

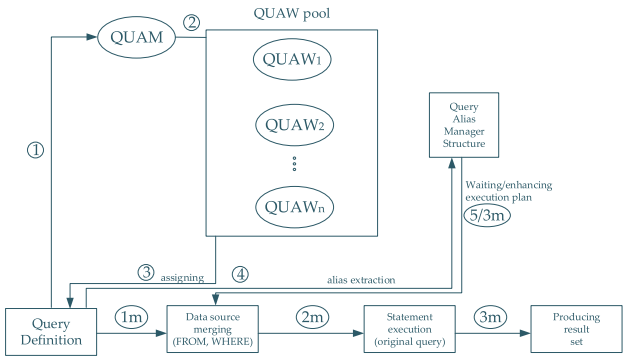


FIGURE 12. Query extraction process flow.

At least one worker process must be present to enable the extraction process. Still, the master process aims to start the extraction immediately by assigning the worker process to the query. Furthermore, multiple workers can be allocated to a single query if there is a nested statement definition. Thus, for each Select clause, one worker can be assigned. The process of query extraction is marked in Fig. 12. If the query is to be executed, the QUAM process is notified by identifying the depth of the query – number of nesting and complexity of the query, followed by the creation of the data structure holding list of extracted aliases and their visibility and applicability. Then, available (free) worker

processes are assigned for each Select clause. Generally, one worker process can be assigned to multiple clauses if not enough workers are available. In that case, nested queries are processed sequentially based on the query definition and references. The created structure supervisor is notified if the particular clause is fully searched. Finally, if the worker finishes their work, its status is changed to available and is autonomously routed to the available worker pool. QUAM process is not notified, limiting its contention (overloading).

As evident, alias extraction and management are done separately, and original statement execution is not influenced. If aliases are present and referenced during the source table merging process, their definitions and mapping would be attempted to be located in the created structure, managed by the master process. If the definition already appeared in it, the processing could continue directly. On the contrary, if the given alias had not yet been defined, the database optimizer would decide and select the next action - either by changing the execution plan or by waiting for the alias to be processed.

Fig. 12 shows the architecture, relevant database objects, processes, and data flow. Designations 1m, 2m, 3m, and 4m refer to the original statement execution flow, while marks 1-5 delimit alias management extension. The structure for holding a list of identified aliases is marked as Query Alias Manager Structure.

Compared to the existing approach, it is important to mention that the original statement is not redefined by applying aliases; instead, they are treated dynamically during the execution of individual clauses, not the whole query.

B. LISTING ALIASES

In the preceding architecture description, Query Alias Manager Structure has been introduced. It is directly associated with the query and, thus, part of the private memory interconnected with the session – Private Global Area (PGA). Whereas it is just devoted to the query definition, it is not shared in any way. Internally, it is organized by the B-tree data structure; the key is the whole function definition in a pure way, or its hash can be used, consisting of the parameter references. It does not use B+tree, while thereś no reason to sort data on the leaf layer in any way.

One of the proposed enhancements is to use multi-index – the first layer is a partition key defined by the function definition. The second layer is a separate index associated with each node of the preceding layer – function definition—the key of the index list of parameters. The architecture of the multi-index function reference is depicted in Fig. 13.

The limitation of the multi-index is just the second layer. Namely, many variants of the function calls can be enhanced by the optional parameters, default values, and overloading in the packages. To ensure general applicability, the following rules are used:

- each function is expanded to its full definition; missing values are replaced by the default values,

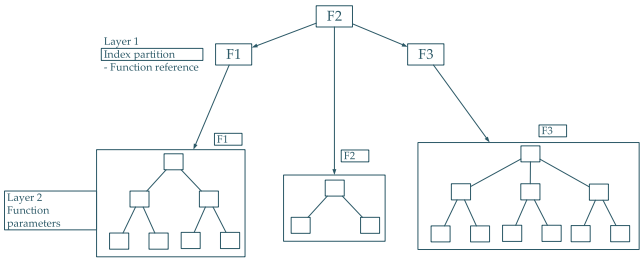


FIGURE 13. Multi-index function reference.

- any overloaded function is ordered and treated as a separate definition (with no relationship to other overloaded functions).

The above rules shape parameters into a common format, which can be treated as an index key.

C. LOCAL QUERY REPOSITORY

The proposed Query Alias Manager Structure is in the instance memory in a private region associated with the session and query. After the statement is done, such a structural reference can be released. In that solution, we generally aim to postpone the release operation to the end of the transaction or session itself (if it does not exceed the assigned PGA capacity). It assumes that a particular query can be later used in the original definition. Thus, the already extracted aliases and references can be used. The same query identification relies on the hash value calculated for each Select statement, which is also used for the execution plan identification and reference-plan hash value.

Query Alias Manager Structure is private and not shared across multiple users or sessions. However, various streams, applications, or logged users can launch the same query. Therefore, one of the proposed enhancements is to move the original Query Alias Manager Structure to the global repository, called Global Query Alias Manager Structure. It is shifted to the shared memory of the instance. It uses a key pair–plan hash value and a set of aliases with their definition as the B-tree. Various techniques were used to limit the capacity of that structure – by number of references, complexity, total costs of the processing, periodicity of the statement call, workload, etc. Based on our other preliminary evaluation study, the most promising solution is just the frequency of calls of the given function. If there are several such functions, then the complexity of the query and the number of extracted aliases is monitored.

The list of extracted aliases is interconnected and shared with the execution plan based on the common Plan hash values (Fig. 14). The pointer direction is from the Global Query Alias Manager Structure to Execution plan storage. This limits dangling pointers in case of releasing memory for the alias references. Moreover, adaptive execution plans can use the same set of aliases for the same query definition.

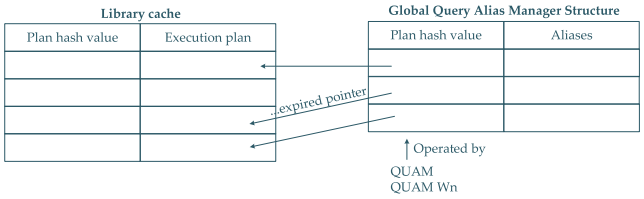


FIGURE 14. Alias extraction management.

D. STRUCTURE SWAPPING

Global Query Alias Manager Structure is located in the instance memory and shared across the query users to serve the global workload. The size of it is dynamic, and the existing Memory Manager background process, which is always active in the database system (forming the core of the instance), sets the right value based on the workload and capacity. However, it is infeasible to expand the size unlimitedly. Sooner or later, it would be necessary to release some stored references and aliases extracted from the query. As stated, several approaches can be used to select the right record to be released from the structure. However, instead of a physical delete operation, it is possible to apply only logical operations. Namely, before the delete, the particular record is stored in the archive repository located in the database. And there, the same rules can be applied to limit the storage capacity, but generally, disc storage has significantly lower prices.

Furthermore, if released in the cloud, various storage types and parameters can be provisioned to balance the performance and costs. Alias Archiver background is, therefore, introduced to ensure records are stored in the database repository before deletion from the global instance memory area. The architecture and record management are shown in Fig. 15.

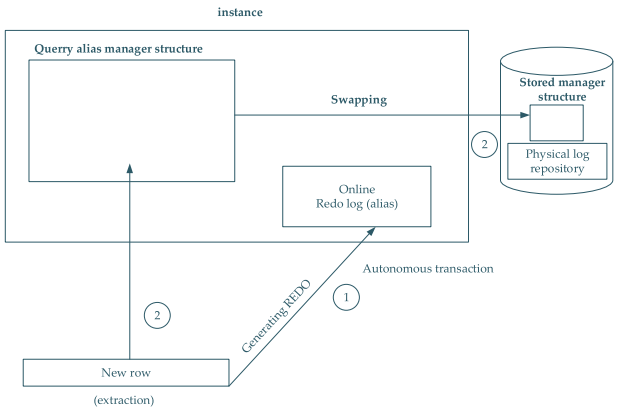


FIGURE 15. Structure swapping.

E. TREATING FUNCTIONS–IDENTIFYING FUNCTIONS, NOT ONLY ALIASES, LOADING DEFINITION

The overall investigation and proposed solutions were primarily inspired by Oracle 23c, enabling reference aliases generally in any clause of the Select statement. The introduced extensions were limited to using aliases extracted before the statement processing. The original query was practically rewritten by replacing the aliases with direct definitions. If the function calls were referenced, instead of calling them multiple times, calculated results were cached, either locally for the query or globally, stored in the Result cache instance memory structure. A different situation, however, occurs if the aliases do not encapsulate the function calls. Precisely, function definitions are not extracted in that case. If the Result cache is not explicitly set for the session or the statement, one function can be called multiple times. To sharpen the problem, if the statement is complex and contains nested queries, one function can be called multiple times. Our proposed solution enhances alias management by function calls. Namely, an alias is automatically created before treating a function, which is then treated in the same manner as described before. Thus, if the alias does not encapsulate a function call, it is created automatically before the statement execution. It consequences in the following facts:

- each function is identified by the alias definition, calculated from the function definition hash,
- each function is called only once,
- statement is rewritten in the initial phases.

The performance of the statement processing is ensured from the perspective of the function calls. However, it has one significant drawback devoted to the third point stated above – the statement is rewritten. The principles are expressed in the following code snippet. The original statement is changed. However, the Hash plan value is calculated from the enhanced query, resulting in processing different inputs. So, multiple Hash plan values can be calculated for one statement because it can be preprocessed to multiple formats and, thus, multiple input shapes. To make the user aware of the consequence of processing and expanding the query through the generation of aliases, a new session (or system) parameter must be introduced:

```
alter {session | system}
 set generate_function_alias =
                          {restricted | ignore | map | force};
```

There are four options for the generate_function_alias parameter:

- *restricted* – additional alias generation for the function calls is restricted thus the statement remains original and is considered as an input for calculating Hash plan value,
- *ignore* – semantics for treating multiple Plan hash values is ignored and the generated alias encapsulates each function call,
- *map* – even though the function aliases can be defined in the original query, they are always replaced by the generic definition obtained by the hash. Thus, the original statement is always mapped to the same format. Consequently, regardless of whether aliases have been defined, the Hash plan value is always the same,

- *force* – the alias definition must explicitly delimit each function.

The principles are expressed in the following code snippet:

```
-- original statement
select count(*), to_char(date_from, 'MMYYYY') date_val
  from data
    group by 2;
-- restricted option
select count(*), to_char(date_from, 'MMYYYY') date_val
  from data
    group by 2;
-- ignore option
select count(*) as num, to_char(date_from, 'MMYYYY') date_val
  from data
    group by date_val;
-- map option
select count(*) as A1, to_char(date_from, 'MMYYYY') as A2
  from data
    group by A2;
-- force option
Exception raised. The expected original format is:
select count(*) as num, to_char(date_from, 'MMYYYY') date_val
  from data
    group by 2;
```

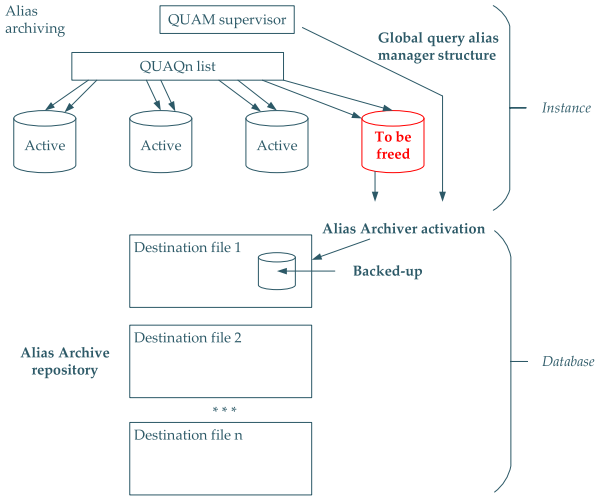


FIGURE 16. Enhanced transaction management.

F. FUNCTION CALL MIGRATIONS

Structures holding the data, extracted aliases, and function references can consist of hundreds of statements, their aliases, various functions, overloading, parameter variants, etc. There can be enormous work done to create, maintain, and fill those structures, either located in the memory or placed at least partially in the database through the alias archiving process. One way or another, securing those data and making them available after the instance restart or any failure is always useful. Although this structure's possible loss or damage will not cause data loss or cannot violate integrity, the costs of re-creating it and restoring the content may be too high. Therefore, the Query Alias Manager Structure can be encapsulated and actively treated within the transactions. However, be aware such transactions are autonomous and do not impact the original data. The transactions automatically record any new row to be loaded into the repository, which can be approved for each row. This would, however, degrade the performance by always reaching transaction ends. Therefore, in bulk, only REDO is generated and shifted to the UNDO tablespace (located in the physical database repository). While the statements do not evolve, there is no update operation for the alias management since it would generate a different execution plan and Plan hash value. Thus, there is no need to reconstruct the historical (original) state, and UNDO's operation can be limited. Thanks to that, after any failure, it is still possible to reconstruct the structure as it existed directly before the failure, but the additional transaction management does not impact on the overall performance.

Furthermore, such transaction logs are grouped together and optionally handled by the log archiving process, just like standard database transactions. The enhanced transaction management for the Query Alias Manager Structure is shown in Fig. 16, reflecting the reduced transaction management by attempting to reach only the REDO structure. REDO generation is in the first phase, then, the new row is stored in the Query Alias Manager Structure. In parallel, the generated log is stored in the database.

Another proposed approach is to refuse the alias management structure's transaction management completely. This brings significant savings in costs and resources. The transactions do not cover changes; thus, there are no additional costs to the database transaction manager. There is no need to generate and save the REDO structure, and the archiving process does not need to be applied. Instead, all changes in the structure are written directly to the database. In principle, it is done as mirroring at the disc level. Thus, even in disc collapse, restoring data from the surviving structure is possible. Moreover, it should be emphasized that the Query Alias Manager Structure does not store any special data, and the entire content can be re-extracted from the source data, although often at a considerable cost.

To ensure data consistency and resistance to change and failures, file versioning on the database layer for the alias management structure is enabled.

The drawback of refusing transactions and managing the data physically in the database lies in the duplicates. A new row is created and stored in the memory, but a copy is stored in the database. From the storage efficiency point of view, the most relevant data is stored and treated twice.

G. RELAXED RULE IN TRANSACTION CORE ENVIRONMENT

As introduced, Query Alias Manager Structure can be in the instance memory, either in a private area or shared across the users. In that case, transaction REDO operations must be present to ensure restoring ability. Otherwise, the whole structure would be lost after the instance failure or generally after the restart of the server. On the other hand, if the transactions do not supervise the management, the whole structure is stored in the database, and I/O loading is necessary. Furthermore, additional demands exist, whereas part of the structure is copied from the database to the instance

memory. Finally, each new row must be primarily stored in the database to make its applicability and restorability after the instance failure.

The proposed intersolution between storing Query Alias Manager Structure only in memory and the physical data layer delimited by the swapping is related to the relaxed rule in the transaction core environment. The data in the memory is not covered by the transactions, but the swapped records are stored physically. Most data are covered and protected in the event of a system failure, whereas they are stored in physical storage. However, which records are stored in the memory that are not protected? Well, the most important records are placed there – the newest records, as well as the most frequently used. Therefore, omitting its protection can cause a loss of performance and relevance.

Such a premise of getting better performance by limiting REDO generation does not bring sufficient power in case of failure.

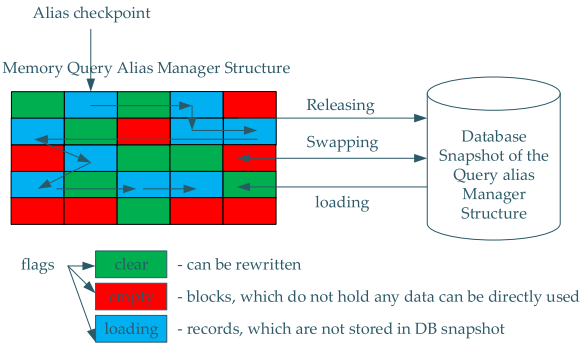


FIGURE 17. Checkpointing.

H. CHECKPOINTING

During the optimization of the Query Alias Manager Structure and swapping between memory and disc storage, we have been investigating optimization techniques to ensure availability and recoverability, but without specific transaction support. The proposed solution introduced in this section is based on memory checkpointing. The proposed architecture is shown in Fig. 17. The memory part of the Query Alias Manager Structure is block-oriented; however, it is a logical concept rather than physical. To be precise, each block takes only one statement extraction. The block contains a flag, which can be clear, empty, or new. Empty blocks are free and can be directly used. They are most preferred for holding new tuples or loading from the storage. Although clear blocks hold the data, they can be directly rewritten since they hold only the data part of the database snapshot. Blocks marked as new hold the data, which needs to be released into the database snapshot. Thus, in case of instance failure, such records would be lost. In principle, these records have been created recently by defining new statements for processing and extracting aliases and functions. To concentrate them by applying them to the new snapshot, those blocks are linked together, forming a linear list. The Alias Checkpointer points to the first occurrence of the new block flag. Some new blocks are copied to the database snapshot in a defined frequency, and particular blocks are marked as clear. Alias Checkpoint is also shifted to the next set of blocks. The introduced Alias Swapper process does this activity. The following parameters influence it:

- *check_freq* – how often the alias checkpoint is invoked, defined by the time frame or dynamic. Time frame frequency is in seconds. Dynamic option means, that the frequency depends on the ratio between clear, empty and new blocks aiming to have free (empty, new) blocks always available.
- *check_ratio* – defines number of blocks to be swapped.

The parameters can be set on the system level by using the alter system command:

```
alter system set check_freq = {value seconds | dynamic};
alter system set check_ratio =
                {number_of_blocks blocks | percentage ratio};
```

The percentage ratio defines the ratio between all new blocks and those that will be backed up (using a swapping operation).

In addition, the checkpoint can be invoked manually:

```
alter system checkpoint aliases;
```

Incorrectly set parameters could cause block contention. In that case, if there is no available free block, even new blocks can be rewritten to serve a new statement. However, it causes performance degradation because for the same query, the already extracted aliases and function call references are lost, and it would be necessary to parse them hard and extract them again in the future execution (at the next processing).

I. PROPOSED SOLUTION SUMMARY

This paper provides several techniques and enhancements to build a robust proposed solution extracting and covering column, expression, and function call aliases. It aims to limit calls to identical functions in different parts of the query, not only one by itself but also reflected by binding to nested and correlated statements. However, more than this alone might be required, as aliases may not delimit specified functions. That's why we introduce our own identifiers and logical alias frames for function calls, thanks to which it is possible to directly reference such functions and process them analogously as if an alias was given. However, naturally, this is only a logical definition, which will not be reflected in the format of the resulting set.

The proposed solution starts with identifying and extracting the aliases placed in the Select clause of the statement, followed by listing aliases. To serve that, a new background master (QUAM) process supervising worker activities (QUAWn) is introduced. The extracted aliases and function calls are placed in an optimized data structure – Query Alias Manager Structure, primarily located in the instance memory. Based on the evaluation study, we concluded that reaching it only in memory is infeasible due to the resource demands

and memory capacity. In contrast, it was shifted from the private area to the shared repository, allowing sharing of the same execution plans and extracted elements. Therefore, a local query repository was introduced, defined by swapping between the instance memory and database storage. Transaction coverage was initially used to ensure consistency, reliability, error-prone, and recoverability after the failure. However, it brought additional processing demands, limited by extracting the alias and function call management to the separate autonomous transactions. Besides, we introduced related transaction rules and alias management checkpointing to ensure performance and robustness. The schema of the whole solution is in Fig. 18. Individual modules were explained in detail in the previous section. The next section provides a computational study of the performance impacts delimited by the costs, storage demands, and processing time.

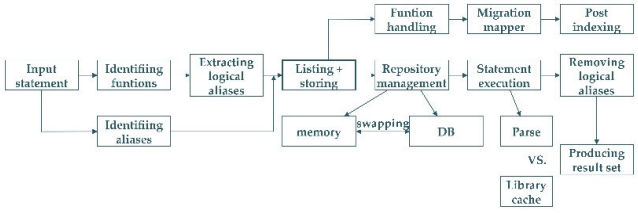


FIGURE 18. Schema of the whole solution.

V. PERFORMANCE EVALUATION STUDY

Several research and performance evaluation studies have been done on the existing solutions. They focus on statement execution, query hash management, and execution plan optimization, followed by storing them in the Library cache of the Shared pool memory structure. Namely, caching results of the function calls can bring benefits if a particular function is executed multiple times with the same parameters by allowing the map function call definition directly to the result set. Based on the evaluation study, automated result caching brings fewer benefits. Instead, complex functions should be extracted by evaluating the processing time and costs of the execution. In [46], function results are delimited by the virtual columns, not stored physically. Function mapping simplifies the whole execution process since the pre-fetched values are associated. Based on the complexity, it can lower the processing time demands by up to 30%. Context switch reduction is discussed in [47] and [48]. Using PRAGMA_UDF optimizing function execution for the SQL usage gets only tiny improvements since the parsed function is in the instance memory for the evaluation [49].

The most promising existing solution relates to the function-based indexes. It is B+tree oriented; the key is the function call itself, enhanced by the various columns. The leaf layer can refer to the results. The structure is composed based on the assumption of the deterministic function reference. The main advantage is directly mapping the function call with the defined parameters.

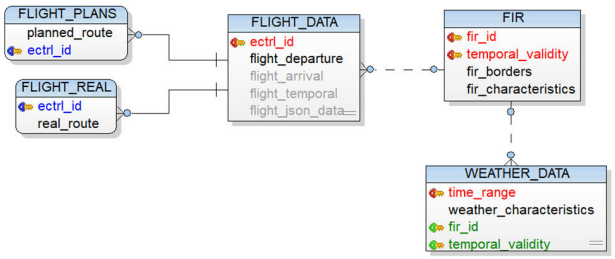


FIGURE 19. Data model.

A dataset getting flight monitoring over Europe was used for this computational evaluation study of the existing approaches. It consisted of 500 000 flights extracted from 2015 to 2018. It stores individual flights, flight positions, flight parameters, flight information region (FIR) assignment, FIR parameters (evolving over time), and weather conditions identifying extreme days. The data model is shown in Fig. 19. Most of the descriptional parameters – flight data, FIR parameters are stored in the JSON format. Compared to the analogous XML definition, a 5% saving is identified on average. It is caused by the necessity of structure and XML schema verification, which is not present in JSON. However, if the JSON structure needs to be supervised by the format definition, an additional 3% were identified. The JSON structure for the flight consisted of 50 parameters, and FIR was delimited by the positional data in an array format plus ten other parameters.

The parameters of the server used for the evaluation are:

- *Processing unit*: AMD Ryzen 5 PRO 5650U, 2.30 GHz, Radeon Graphics
- *Memory*: Kingston, DDR4 type, 2 × 32GB, 3200MHz, CL20
- *Storage*: 2TB, NVMe disc type, PCIe Gen3 × 4, 3500MB/s for read/write operations
- *Operating system*: Windows Server 2022, x64
- *Database system*: Oracle Database 21c Enterprise Edition Release 21.3.0.0.0 Production Version.

Tab. 1 provides a summary of the existing solutions, dealing with the total costs of the processing. Whereas the absolute values do not provide sufficient performance image, caused by the versioning, memory structure references, and resource competitions, values are expressed in percentage to declare the differences. The core solution with no enhancements is treated as a reference, providing 100%. Tab. 1 deals with the virtual columns, referencing function results in a specific database repository located in the database, context switches, and function-based indexes. The depicted results are related to the airplane monitoring during the flight. Individual date and time elements are extracted from the definition and are treated separately on the second granularity. The stated solutions are used as a benchmark for the proposed solutions consideration. Calculated costs form the metrics for evaluating the quality of the solution. They are provided during the execution of the statement. Estimated costs are also used for comparing various execution plans,

to identify the best suitable option. The optimization of the costs stated in the Tab. 1 is based on enhancing function access through the indexes and pre-storing results in the Result cache memory structure.

TABLE 1. Results – existing approaches.

	Costs [%]	Optimized costs [%]
Original solution	100	
Result cache	80.25	64.52
Virtual column	79.61	
Specific function database repository	61.62	
Context switches reduction	97.99	
Function-based index	55.12	50.47
Implicit function mapping	92.01	

The results show two competitive streams – Result cache and Virtual column. The result cache stores multiple functions temporarily and reflects the various parameters and mapped results. Thus, not all combinations of the parameters used for the particular function are stored. On the other hand, the virtual column takes all the occurrences but only one function. It is impossible to combine multiple functions, while virtual columns always relate to one function only. There is only a tiny difference between those two solutions – 0.64%. However, by combining those two solutions and referencing function values by the virtual columns, which can be directly obtained using the Result cache, total cost demands are reduced by more than 12% to 64.52%. By enhancing the environment to store the whole function set reference in the specific database repository, the total cost demands are 61.62%. Compared to the original solution, it refers to the improvement by 38.38%. It is, however, worth mentioning that all function references are stored, even if they are no longer used later. This results in significant demands on disk space, database, and I/O operations. Based on our computational study, the total requirements are more than 10-fold for flight monitoring, date extraction, and tracking of aircraft parameters. If the storage is reduced to at least three uses, the costs are 62.74%, which is still better than combining the Result cache and virtual column. The difference (1.12%) refers to the additional requirements to execute a function with the same parameters multiple times.

Context switch reduction provides only a tiny improvement. Namely, in our environment, there is only a small set of used functions, which differ in complexity and used parameters. Mapping the definition to the memory and dynamic SQL reflection can bring power and reliably reduce the impact of the context switch.

The function-based index obtained the best solution, while it is not a flat structure but rather a B+tree. Searching across the set is far more effective than the virtual column or pure Result cache. By enhancing it with the Result cache, processing costs are dropped from 55.12% to 50.47%. Furthermore, if it is a memory Buffer cache located, demands can be lowered to 46.01%. However, this approach's limitation is related to the reliability of the processing and data type mapping. Namely, it requires precise data type specification on both sites of the condition - both data types must be the same to ensure proper performance. If not, an implicit conversion is automatically applied behind the scenes to serve the execution frame. It, however, means that the original function-based index cannot be used since the conversion function should enhance it. The total processing costs rise to 92.01%, which reflects a significant increase compared to the original function index, which requires 55.12% and 50.47%, respectively. The index cannot be used; if enabled, the definition and processing can relate only to storing values in the Result cache. A specific enhancement can be applied if the condition consists of the function on one side and the second side is not covered by the function (referring to the constant or ordinary table column); then, the function can remain original, and the conversion function is done on the ordinary value. In that case, it can be enhanced by the non-convert function hint. Let's consider the following example:

```
select /*+no-convert*/
 from ...
     where to_char(date_from, 'MM')=12;
```

The left part of the condition produces a character string, while the numerical representation denotes the right side. The no-convert hint navigates the system to convert the original column value, not to embed function call in the function call:

```
to_number(to_char(date_from, 'MM'))=12;        -- is refused
to_char(date_from, 'MM')=to_char(12);          -- is applied
```

The function-based index can continue to be used by applying a no-convert hint. Furthermore, the conversion function can be treated as a common function and refer to the Result cache. Consequently, the additional processing cost demands range from 2% to 3%, depending on the original and destination data types.

The above experiment aims to monitor the whole flight, getting its parameters and FIR assignment. There are various functions to sharpen the performance – getting date and time elements, getting flight data operated by the function call transforming JSON into a relational form (textual representation), and checking weather impacts. The function obtains all those property calls:

```
select /*+no-convert*/
            to_char(flight_departure,
                                'DD.MM.YYYY HH24:MI:SS'),
            to_char(flight_arrival, 'DD.MM.YYYY HH24:MI:SS'),
            extract(hour from flight_temporal),
            extract(minute from flight_temporal),
            extract(second from flight_temporal),
            monitor_flight(flight_JSON_data, ECTRL_ID),
            get_FIR_assignment(ECTRL_ID, flight_temporal)
  from flight_data
             join FIR on(fir_borders(lon, lat)=fir_borders)
             join weather_data(time_range)
    where to_char(flight_departure, 'YYYY')=2018;
```

The second evaluation stream deals with the existing alias management and provides enhancements in Oracle 23c. It uses five solutions for referring function calls and alias

references. The REF solution does not consider aliases, and the particular functions are referenced multiple times, enhanced by the Result cache, so the function is called for the defined attribute set just once. SOL1 uses statement nesting by calculating the function outputs in the first step, followed by their direct usage in the outer query. Thus, individual clauses of the outer query refer to the direct values like ordinary attributes obtained from the database. SOL2 is based on pre-processing using dynamic view (with clause extensions introduced in Oracle Database 12c Release allowing subquery factoring clause enhancements, like subquery factoring, deterministic keyword hint, or PRAGMA UDF). SOL3 uses a materialized view for each table. The materialized view logs incrementally (fast) refresh them. SOL4 uses alias usage enhancements introduced in Oracle Database 23c, offering to refer to the defined aliases in any clause of the same statement. Tab. 2 shows the processing time results and the implicit conversions' impact on character strings and numerical values. The implicit conversion impacts range from 6.28% to 13.47%:

TABLE 2. Results – implicit conversion impacts.

		REF	SOL1	SOL2	SOL3	SOL4
	Costs	3660 (24% CPU)	3660 (24% CPU)	3660 (24% CPU)	2562 (16% CPU)	3660 (24% CPU)
	Processing time [ss.ff]	23.25	27.33	21.78	14.46	18.24
To_number implicit conversion	Processing time [ss.ff]	24.71	29.36	23.76	15.90	19.74
To_char implicit conversion	Processing time [ss.ff]	25.87	31.01	23.96	16.21	20.06

Existing solutions provide a relevant performance layer for data management and function references. Most of the described solutions improve performance in processing time drops, except for the SOL1, based on the statement nesting. First, it requires parsing, processing, handling, and evaluating additional queries. However, the most significant drawback of this approach is extracting the Where condition to the separate processing phase. Therefore, the data reduction factor cannot be immediately applied, and huge data sets need to be processed, even in case of later refusal by the conditional processing. SOL2 and SOL3 use views. The best solution relates to the materialized view, in which the function results are already calculated. However, it applies only to the data retrieval process (Select statement). For any change operations, the materialized view log for the table must be filled to ensure the data correctness. Furthermore, the man transaction processing the data must also supervise the fast refresh of the materialized view itself. SOL4 expands the SQL language principles by offering alias referencing in any statement clause. There is no necessity for query nesting, nor pre-calculations. Instead, the alias definitions are extracted before the statement processing by associating it initially with the flat table structure.

The next computational study evaluation part focuses on the performance of the proposed solutions, emphasizing two aspects – monitoring the whole flight from the departure to the landing, taxi, and parking position on one side (ES1) and getting a snapshot of the aircraft positions at a defined time on the other side (ES2). It also aims to get the flight parameters, status, FIR assignment, and weather situation. The used data set and environment parameters remain the same.

Tab. 3 shows the results of the ES1 phase by emphasizing the main profiles of the proposed solutions. Each evaluation was done ten times for 100 flights. The results express the average values. The reference solution was considered the extended alias management introduced in Oracle Database 23c, enhanced by the Result caching 20% of the most often called functions. By reflecting on our previous preliminary results, it seems that adding a higher percentage for storing results does not bring any significant benefit because of the size demands (10% of Result caching requires approximately 9.5GB) because of the data description complexity provided as an output. Too high value for the Result caching soon hits the capacity of the whole memory. However, the Result cache of the Shared pool memory structure is just a small part of the required instance memory. Since its size can be set dynamically, it results in limiting Buffer cache, so the function calls are pre-fetched. Still, the ordinary data inputs are not forming the system's bottleneck. The individual parts of the evaluated solutions follow each other; that is, each subsequent module takes the properties of the previous one, as mentioned in the description of the proposed solutions and concepts of improving performance, availability, and reachability. There are four stages for the evaluation:

- *stage 1* – identifying, extracting and listing aliases,
- *stage 2* – local query repository,
- *stage 3* – swapped alias and function extration structure between instance memory and database, operated by the QUAM and its workers. The whole structure is shared across the sessions in a whole instance.
- *stage 4* – enhancing the system by the virtual logical aliases treating functions.

TABLE 3. Results – ES1 performance results - stages.

ES1	REFERENCE SOLUTION (Oracle 23c)	Stage 1	Stage 2	Stage 3	Stage 4
Processing time [ss.ff]	33.12	31.77	31.45	27.01	22.07
Processing time [%]	100	95.92	94.96	81.55	66.64

ES2 extracted the same number of flights (100) at the defined times (10). Thus, the same data portion compared to ES1 was processed, the same number of the data, however, treated in a different format and shape. As evident from Tab. 4, processing time demands are almost the same, the

differences between individual techniques and stages are analogous.

TABLE 4. Results – ES2 performance results - stages.

ES2	REFERENCE SOLUTION (Oracle 23c)	Stage 1	Stage 2	Stage 3	Stage 4
Processing time [ss.ff]	29.12	27.09	26.93	23.26	18.42
Processing time [%]	100	93.03	92.48	79.87	63.26

Stage 1 is the initial phase for the storage repository extracting, listing, and maintaining aliase, getting an improvement of 4% for ES1 and almost 7% for ES2. Storing data locally does not bring reliable benefits since the same queries, execution plans, and functions can be shared across multiple users. A significant processing cost drop is reflected in stage 3, defined by the swapping. Namely, for ES1, it refers to the 13.41% compared to stage 2. Globally, it represents a processing cost drop of 18.45%. Analogously, for ES2 it refers to the 12.61% processing cost drop compared to stage 2. Reflecting the reference solution introduced in Oracle 23c, it gets a drop of 20.13%. The last stage is the most promising in case the alias definitions do not encapsulate the functions. Based on the evaluated environment, Virtual logical aliases can improve up to 33.36% for ES1 and 36.74% for ES2. Graphical representation of the reached results is depicted in Fig. 20.

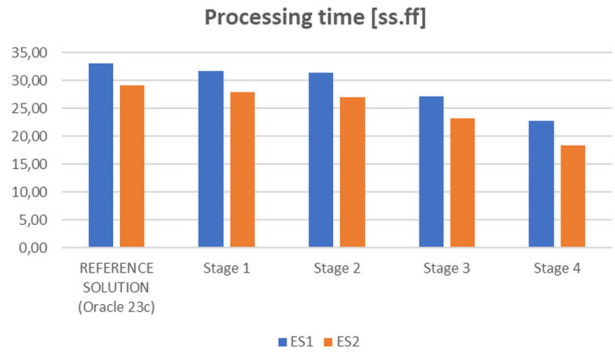


FIGURE 20. Performance evaluation results.

The limitation of the above-proposed solutions lies in reliability and consistency. Namely, if the function is shared across the whole instance, it must be always checked the reference and owner of that function to ensure the correct call is performed. Owner reference is always stored in the proposed data structure to serve that. However, more than that would be needed since the function header can be enhanced by the authorization scheme definition, which is not directly present when parsing the query. Thatś one of the streams to be considered in the future research and development.

There are a bunch of streams, possibilities, and directions within the proposed research.

There are many future strategies to investigate, like the impact of checkpointing, setting the right rules to minimize possible function references after the instance restart, ensuring proper performance, and shifting between the instance memory and physical storage. Further investigation will also be done on stage 2, which needs more improvements. However, when dealing with multiple authorization schemes, combining shared structure properties and local query reflections would be necessary. Another stream points to the scalability of the whole solution by creating data partitions. Finally, we aim to investigate data distribution techniques across multiple servers, calling remote functions, etc. In that distributed environment, we would like to place our alias and function extraction techniques to pre-fetch the results and investigate two parts of the performance impacts – executing remote functionalities and distributing created data structures across the servers by aiming to create a general methodology of sharing and distributing.

VI. CONCLUSION

Complex data analytics is an inseparable part of current information technology. Data warehouses, marts, lakes, or any other variants are created to serve the environment for data management and storage. Autonomous Oracle Database in the cloud environment even sharpens the problem by making it available and easily reachable. Inside the analytical-oriented environment, complex queries are present, which need to be optimized to ensure performance and limit used resources. Oracle database system provides one of the most performant solutions and many technical enhancements. This paper deals with the function and alias references in the Select statement. Currently, the Oracle 23c version goes beyond the SQL standardization and allows the usage of the defined aliases at any query level. This, naturally, requires changes at the query processing stage level and the creation of the framework for identifying, treating, and referencing aliases to make them applicable. Furthermore, behind the scenes, it causes the query transformation to refer to the definition associated with the aliases. This can have a significant impact on the performance and function references.

This paper summarizes existing techniques and approaches , focusing on the function performance, context switches between SQL language and procedural (PL/SQL) interface, virtual columns holding function results, materialized views, and functions covered by the index layer. Besides, column aliases and positional notation management proposed in version 23c are discussed, making the user query definition far easier. On the other hand, as stated, parsing and query evaluation phases must be enhanced. These solutions are also discussed in the performance evaluation section, serving as the reference layer for our proposed solutions reflection.

Architectures are methods when dealing with the proposed techniques, and they are always modular solutions with multiple optimization stages. Firstly, column aliases are introduced and extracted, followed by optimizing the access layer by making the alias references. Such a layer was

primarily stored locally and associated with one query in a private session area. However, in a practical environment, multiple users and sessions can attempt to execute the same source codes, same queries, and call the same functions, so it is relevant to share the content. This makes the structure more complex and more storage-demanding since it is not later associated with only one query duration. Therefore, techniques for swapping data between the memory and database storage layer were proposed to ensure data availability and limit storage demands and I/O operations. The key concept proposed in this paper deals with function management generally, delimited not only by the specified aliases. Thus, virtual logical aliases are introduced, allowing to reference functions. By applying Result caching, additional memory, and storage structures operated by the introduced master process supervising worker activities, overall costs can be dropped to less than 67%, based on the workload activity. This paper used a positional flight dataset to monitor the flight and FIR assignment. Based on the results, the provided solutions bring a strong additional power by reducing processing time and costs. However, they require additional storage capacity and shifting between the database and memory. However, in the currently widespread cloud environment, the main focus is on the performance and demands of the complex queries, mostly expressed by the processing time. Individual storage resources are now cheap and easily provisionable.

Existing approaches, developments in the field, and proposed extensions in this paper open up further possibilities for research and performance optimization. During the project's next phases, we will emphasize priority handling, so not all extracted aliases and function references will be stored. Instead, there will be a set of rules to ensure accessibility on one side and limit storage demands on the other. This can invoke a release rule to manage the data structure based on the current capacity and extractability optimization. Another research stream relates to migration techniques, exports, and correct mapping. Although some research strategies have been already done in this area, and we also partially address the issue in this paper, there are still a lot of possibilities for improvement.

ACKNOWLEDGMENT

Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the Slovak Academic Association for International Cooperation (SAAIC). Neither the European Union nor SAAIC can be held responsible for them.

REFERENCES

[1] H. Cai, Z. Qian, and J. Jiang, "Application of association rule algorithm in distributed new SQL database design," in *Proc. IEEE Int. Conf. Integr. Circuits Commun. Syst. (ICICACS)*, Feb. 2023, pp. 1–5, doi: 10.1109/ICICACS57338.2023.10100098.

[2] S. Pendse, "Oracle database in-memory on active data guard: Real-time analytics on a standby database," in *Proc. Int. Conf. Data Eng.*, Apr. 2020, pp. 1570–1578, doi: 10.1109/ICDE48307.2020.00139.

[3] J. Janàček and M. Kvet, "Adaptive parameter setting for public service system design," in *Proc. 15th Int. Conf. Strategic Manage. Support Inf. Syst. (SMSIS)*, Jan. 2023, pp. 161–168.

[4] G. Rani, T. Sharma, and A. Sharma, "Future database technologies for big data analytics," in *Proc. Int. Conf. Intell. Syst. for Commun., IoT Secur. (ICISCoIS)*, Feb. 2023, pp. 349–354, doi: 10.1109/ICISCoIS56541.2023.10100525.

[5] R. Chauhan and E. Yafi, "Big data analytics for prediction modelling in healthcare databases," in *Proc. 15th Int. Conf. Ubiquitous Inf. Manage. Commun. (IMCOM)*, Jan. 2021, pp. 1–5, doi: 10.1109/IMCOM51814.2021.9377403.

[6] Q. He, F. Zhang, G. Bian, W. Zhang, D. Duan, Z. Li, and C. Chen, "Research on data routing strategy of deduplication in cloud environment," *IEEE Access*, vol. 10, pp. 9529–9542, 2022, doi: 10.1109/ACCESS.2021.3139757.

[7] E. Roske, T. McMullen, and G. Schwartzberg, *Look Smarter Than You Are With Oracle Analytics Cloud Standard Edition*. NC, USA: Lulu Press, 2017, p. 740.

[8] S. V. Oprea, A. Bara, V. Diaconita, C. Ceaparu, and A. A. Ducman, "Big data processing for commercial buildings and assessing flexibility in the context of citizen energy communities," *IEEE Access*, vol. 9, pp. 168715–168730, 2021, doi: 10.1109/ACCESS.2021.3137352.

[9] S. M. N. Mustafa, M. U. Farooq, S. S. Zehra, and J. P. T. Noronha, "A comparative study of the performance of real time databases and big data analytics frameworks," in *Proc. 7th Int. Multi-Topic ICT Conf. (IMTIC)*, May 2023, pp. 1–7, doi: 10.1109/IMTIC58887.2023.10178651.

[10] D. Kuhn and T. Kyte, *Expert Oracle Database Architecture*. New York, NY, USA: Apress, 2021.

[11] M. Kvet, "Enhanced data locking to serve ACID transaction properties in the Oracle database," in *Proc. 34th Conf. Open Innov. Assoc. (FRUCT)*, pp. 73–80. Accessed: Dec. 13, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10328165

[12] M. Kvet, "Developing robust date and time oriented applications in Oracle cloud a comprehensive guide to efficient data and time management in Oracle cloud," Packt Publishing, Tech. Rep., 2023, p. 464.

[13] H. Liu, Q. Chen, N. Pan, Y. Sun, Y. An, and D. Pan, "UAV stocktaking task-planning for industrial warehouses based on the improved hybrid differential evolution algorithm," *IEEE Trans. Ind. Informat.*, vol. 18, no. 1, pp. 582–591, Jan. 2022, doi: 10.1109/TII.2021.3054172.

[14] D. Kuhn and T. Kyte, *Oracle Database Transactions and Locking Revealed: Building High Performance Through Concurrency*. New York, NY, USA: Apress, 2020.

[15] B. M. Sharma, K. M. Krishnakumar, and R. Panda, "Oracle autonomous database in enterprise architecture: Utilize Oracle cloud infrastructure autonomous databases for better consolidation, automation and security," Packt Publishing, 2022.

[16] *SQL 2003 Standard Support in Oracle Database 10G*, 2003. [Online]. Available: https://www.oracle.com/technetwork/database/sql-2003-twp-129141.pdf

[17] *Oracle-Base—Oracle 23c Articles*. Accessed: Dec. 13, 2023. [Online]. Available: https://oracle-base.com/articles/23c/articles-23c

[18] *Database 23c | Oracle*. Accessed: Dec. 13, 2023. [Online]. Available: https://www.oracle.com/database/23c/

[19] *Oracle Database 23c: The Next Long Term Support Release*. Accessed: Dec. 13, 2023. [Online]. Available: https://blogs.oracle.com/database/post/oracle-database-23c-the-next-long-term-support-release

[20] R. Cornejo, "Dynamic Oracle performance analytics: using normalized metrics to improve database speed," Apress, Tech. Rep., 2018.

[21] J. Li and J. Wang, "Index design of electronic medical record database using blockchain," in *Proc. 5th Int. Conf. Mech., Control Comput. Eng. (ICMCCE)*, Dec. 2020, pp. 2003–2008, doi: 10.1109/ICMCCE51767.2020.00438.

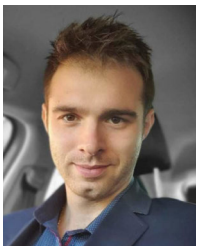
[22] L. Bulysheva, A. Bulyshev, and M. Kataev, "Visual database design: Indexing methods," in *Proc. 6th Int. Conf. Enterprise Syst. (ES)*, Oct. 2018, pp. 25–29, doi: 10.1109/ES.2018.00011.

[23] R. Yadav, S. R. Valluri, and M. Zaït, "AIM: A practical approach to automated index management for SQL databases," in *Proc. IEEE 39th Int. Conf. Data Eng. (ICDE)*, Apr. 2023, pp. 3349–3362, doi: 10.1109/icde55515.2023.00257.

[24] M. M. Rovnyagin, S. O. Dmitriev, A. S. Hrapov, A. A. Maksutov, and I. A. Turovskiy, "Database storage format for high performance analytics of immutable data," in *Proc. IEEE Conf. Russian Young Researchers Electr. Electron. Eng. (ElConRus)*, Jan. 2021, pp. 618–622, doi: 10.1109/ElConRus51938.2021.9396453.

[25] G. Arora, S. Kalra, A. Bhatia, and K. Tiwari, "PalmHashNet: Palmprint hashing network for indexing large databases to boost identification," *IEEE Access*, vol. 9, pp. 145912–145928, 2021, doi: 10.1109/ACCESS.2021.3123291.
[26] Y. Zhou, Z. Chen, and K. Li, "Second-level buffer cache management," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 505–519, Jun. 2004, doi: 10.1109/TPDS.2004.13.
[27] B. Rosenzweig and E. S. Rakhimov, "Oracle PL/SQL by example," Oracle Press, Tech. Rep., 2023, p. 480.
[28] I. Stanoi, C. A. Lang, and S. Padmanabhan, "Hint and run: Accelerating XPath queries," in *Proc. 9th Int. Database Eng. Appl. Symp. (IDEAS05)*, 2005, pp. 253–262, doi: 10.1109/ideas.2005.33.
[29] M. Hassan, R. Aihajj, M. J. Ridley, and K. Barker, "Database selection and keyword search of structured databases: Powerful search for naive users," in *Proc. IEEE Int. Conf. Inf. Reuse Integr. (IRI)*, Jul. 2003, pp. 175–182, doi: 10.1109/IRI.2003.1251411.
[30] A. Gupta and I. S. Mumick, *Materialized Views: Techniques, Implementations, and Applications*. Cambridge, U.K.: MIT Press, 1999, p. 589.
[31] Alka and A. Gosain, "A comparative study of materialised view selection in data warehouse environment," in *Proc. 5th Int. Conf. Comput. Intell. Commun. Netw.*, Sep. 2013, pp. 455–459, doi: 10.1109/CICN.2013.100.
[32] S. U. Khan, "Data warehouse enhancement manipulating materialized view hierarchy," in *Proc. 8th Int. Conf. Digit. Inf. Manage. (ICDIM)*, Sep. 2013, pp. 369–372, doi: 10.1109/ICDIM.2013.6694037.
[33] S. Kurzadkar and A. Bajpayee, "Anatomization of miscellaneous approaches for selection and maintenance of materialized view," in *Proc. IEEE 9th Int. Conf. Intell. Syst. Control (ISCO)*, Jan. 2015, pp. 1–5, doi: 10.1109/ISCO.2015.7282236.
[34] M. B. Roeser, "Create materialized view log," Oracle Press, Tech. Rep., 2012.
[35] M. Kvet, "Dangling predicates and function call optimization in the Oracle database," in *Proc. Commun. Inf. Technol. (KIT)*, Oct. 2023, pp. 70–79, doi: 10.1109/kit59097.2023.10297114.
[36] M. Kvet and J. Papan, "The complexity of the data retrieval process using the proposed index extension," *IEEE Access*, vol. 10, pp. 46187–46213, 2022, doi: 10.1109/ACCESS.2022.3170711.
[37] R. Cornejo, *Dynamic Oracle Performance Analytics: Using Normalized Metrics to Improve Database Speed*. Apress, 2018.
[38] M. H. Durneková and M. Kvet, "Optimization of the SELECT statement containing window functions," in *Proc. Int. Conf. Inf. Digit. Technol. (IDT)*, Jun. 2023, pp. 267–272, doi: 10.1109/idt59031.2023.10194457.
[39] M. Yu, C. Chai, and G. Yu, "A tree-based indexing approach for diverse textual similarity search," *IEEE Access*, vol. 9, pp. 8866–8876, 2021, doi: 10.1109/ACCESS.2020.3022057.
[40] S. Zhang, S. Ray, R. Lu, and Y. Zheng, "Efficient learned spatial index with interpolation function based learned model," *IEEE Trans. Big Data*, vol. 9, no. 2, pp. 733–745, Apr. 2023, doi: 10.1109/TBDATA.2022.3186857.
[41] M. Kvet, "Identifying, managing, and accessing undefined tuple states in relational databases," in *Proc. Int. Conf. Smart Syst. Technol. (SST)*, Oct. 2022, pp. 165–172, doi: 10.1109/sst55530.2022.9954691.
[42] M. Kvet, "Referencing validity assignment using B+tree index enhancements," in *Proc. World Symp. Digital Intelligence Syst. Mach. (DISA)*, Nov. 2023, pp. 145–153, doi: 10.1109/DISA59116.2023.10308931.
[43] M. Malcher and D. Kuhn, *Pro Oracle Database 23c Administration: Manage and Safeguard Your Organization's Data*. Apress, 2024, p. 588.
[44] A. Agrawal, *Oracle Database Database Administrator's Guide, 23c*. USA: Oracle Press, 1996.
[45] *ORACLE-BASE—GROUP BY and HAVING Clauses Using Column Aliases in Oracle Database 23c*. Accessed: Dec. 13, 2023. [Online]. Available: https://oracle-base.com/articles/23c/group-by-and-having-clause-using-column-alias-or-column-position-23c
[46] E. I. Chong, M. Perry, and S. Das, "Improving RDF query performance using in-memory virtual columns in oracle database," *Proc. Int. Conf. Data Eng.*, vol. 2019-April, pp. 1814–1819, Apr. 2019, doi: 10.1109/ICDE.2019.00197.
[47] M. Kersten, Y. Zhang, P. Katsogridakis, P. Koutsourakis, and J. van Ruth, "Database resource allocation based on resilient intermediates," in *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci. (CloudCom)*, Dec. 2018, pp. 314–319, doi: 10.1109/CloudCom2018.2018.00067.
[48] M. Kvet, "Impact of disc types on database performance," in *Proc. IEEE 16th Int. Sci. Conf. Informat. (Informatics)*, Nov. 2022, pp. 188–195, doi: 10.1109/Informatics57926.2022.10083421.
[49] W. Li, N. Li, J. Yan, Z. Zhang, P. Yu, and G. Long, "Secure and high-quality watermarking algorithms for relational database based on semantic," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 7, pp. 7440–7456, Jul. 2022, doi: 10.1109/TKDE.2022.3194191.

MICHAL KVET (Member, IEEE) became an Associate Professor in applied informatics with the Faculty of Management Science and Informatics, University of Žilina, Slovakia, in 2020. He is currently a recognized Researcher, a Conference Speaker, and an Oracle ACE Alumn. He is the author of several text-books and monography in temporal database processing. He is the author of more than 70 scientific articles indexed in IEEE-Xplore, Scopus or WOS. He is certified for SQL, PL/SQL, analytics, and cloud databases. His research is devoted to the temporal databases, indexing, performance, analytics, and cloud computing. He strongly participates with Oracle Academy and he is a part of multiple Erasmus+ projects. Besides, he is a Consortium Leader of the Erasmus+ project dealing with the environmental analytics. He also organizes multiple database workshops annually.



JOZEF PAPAN received the dual Ph.D. degree in applied informatics from the Faculty of Management Science and Informatics, University of Žilina, Slovakia, in 2015 and 2020, respectively.

He is currently the Head of the IP Fast Reroute Research Team, the Director of the Fortinet Network Security Academy, and a member of Cisco Academy with the Faculty of Management Science and Informatics. He is the author or coauthor of more than 30 scientific papers published in scientific journals and presented at international conferences. His research interests include IP fast reroute, fault-tolerance, protocols and services in IP networks, WSN, the IoT, the modeling and simulation of computer networks, smart sensors, wireless technology, portable devices, technical cybernetics, and cloud computing. He is the Teacher of the following subjects: Securing Networks with Fortinet (Fortinet Academy), Principles of ICS (Cisco), and Network Architectures (Linux + Networks).