Data Analysis - Aggregate Function Performance

Michal Kvet
Department of Informatics
Žilinská unvierzita v Žiline, Fakulta riadenia a informatiky
Žilina, Slovakia
michal.kvet@uniza.sk

***Abstract*— Increasingly expanding data analytics brings the need to store a large and structurally diverse amount of data. Related to this is the design of analytical data structures and performance optimization. Individual data are typically aggregated in reports. The goal of this paper is to create a methodology for using aggregation functions in analytical-transactional SQL databases. We focus on the impact of the used parameter format and values, as well as grant total calculation technique with the aim of minimizing processing costs and time.**

Keywords—environmental data processing, transport systems, data analysis, aggregate functions, SQL, temporal databases

I. INTRODUCTION

In order to be able to make a qualified decision, it is essential to surround yourself with data. Of course, it is important that these data are correct, consistent and verified [7]. There are a number of techniques, mainly at the level of statistics and data characteristics, by means of which it is possible to ensure the quality of the input data, which are subsequently stored in the database. This process is typically secured through transactions with 4 basic properties (*ACID - atomicity, consistency, isolation* and *durability*). In the past, transactional data focused on currently valid data and the entire system was conventional. This meant that the currently valid rows were overwritten in the event of a change and the original values were not preserved [5]. They could only be partially identified through transaction logs and archives, which were quite difficult to search in. Moreover, they contained only a limited number of data, specifically only small portion of the the last changes. Also, there were many control commands in these logs and therefore the processing efficiency was slow and resource-demanding [6].

As stated, data are needed to ensure correct and qualified decisions. Data-driven decision making is an inseparable part of any intelligent information system. Such decisions are supported by the past and reflect individual changes. They bring a competitive advantage.

It is therefore obvious that we need to store not only currently valid states, but also states valid in the past and also plans of states valid in the future. This is the only way we get a comprehensive view of the data, development of changes and reflection [4] [5]. A prerequisite for data analytics are temporal databases, which demarcate individual records with time stamps. They are usually values expressing validity forming a uni-temporal system (containing one time dimension). If we add another temporal reflection, a bi-temporal system is created, which is characterized precisely by the extension of transactional temporal reflection. It offers the possibility to store not only changes over time, but for each change it is also possible to record possible corrections, delayed data, etc. This system is mainly used in communication systems, where delays or data value corrections can occur. But at the same time, all values must be stored in the entire time spectrum, since even in this way incorrect values could be used for the calculation of reports, analytics and aggregations. A typical example where bi-temporal systems are used is transportation - obtaining values from sensors, communication between vehicles and ad-hoc networks between vehicles and the infrastructure itself [3]. Another example is depicted by the environmental data and analyzing complex data images [2] [8] [9] [10].

Fig. 1 shows the concept of temporal databases from the perspective of dimension processing. The conventional system stores only currently valid states. The uni-temporal model focuses on validity, while the multi-temporal system provides a universal solution with the possibility of modeling an unlimited amount of time dimensions. Note that all dimensions are part of the data identifier itself, and thus one object can be defined by several states, but these states must be disjoint within time dimensions. Such a requirement can be ensured either by the structural data model itself, or by triggers – procedural language in general.

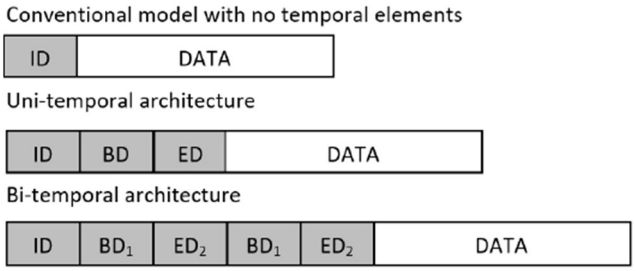

Fig. 1. Temporal data modeling [5]

Fig. 1 is based on the object-level temporal architecture, where the entire record as such is bounded by validity. Thus, if any change occurs, regardless of the individual attributes and their values, a completely new image is created. It can generate a significant amount of duplicates. Federated object-level temporal model provides a partial solution, in which individual attributes are categorized according to the frequency of changes and the original table is divided into federated partitions.

Attribute-oriented temporal approach [6] focuses on the attributes, which are encapsulated separately by the temporal spheres. Thus, valid states is created as a composition of indivudula attributes valid at the defined timepoint.

A hybrid model was introduced in [5] and is based on creating synchronization groups composed, restructured and dropped dynamically.

Attribute and group-level temporal models are rather logical, physically, several layers supervised by the background processes are present to serve the management, mapping and state composition [5]. Fig. 2 shows the group-level temporal architecture as the most generalized solution. It consists of multiple layers. The currently valid states are stored in the first most uppermost layer. They are modeled at

the level of the objects. This concept is applied in most temporal systems [1] so that it is possible to access the currently valid states directly using the data connector, and thus the existing conventional systems will not need structural changes. Temporal layer, as a heart of the system is stored in the second layer, responsible for providing historical data or future plans, as well as reports and change monitoring. Historical and future valid data references are present in the third layer, formed in forms of table blocks. Last three layers are responsible for synchronization group management, composition, restructuralization, etc.

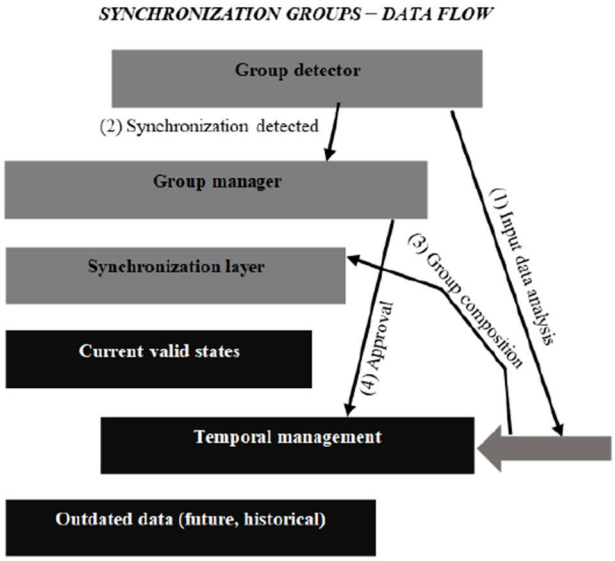

Fig. 2. Group level temporal model [5]

All these models are primarily intended for transaction-oriented systems, which are served as a data source for the analytical oriented architectures and repositories, however, temporal systems can be easily applied in the analytics as the dimensions, referencing multi-temporal architectures in a general level.

When working with complex data and analytics, it is necessary to aggregate data and solve complex calculations. Morever, in the case of temporal systems, these calculations are even enhanced by the time slots [1].

Aggregate functions return a single row as a result for each defined group of rows. Thus, the number of produced rows can be reduced. They commonly use *Group By* clause in the Select statement to define the group for which the aggregate function is calculated. Aggregate functions can be located in the *Select* and *Having* clauses, just in case the condition needs to be based on the aggregate function.

This paper provides a methodology of aggregate function definition in terms of getting proper performance. Precisely, most of the analytical systems do not rely on the performance of the aggregate functions, they simply point to the produced data. The impacts of the definition is part of this paper contribution.

The paper is organized as follows: Section 2 deals with the principles of aggregate functions. Section 3 points to the parameters and mapping impacts.. Performance evaluation study is in section 4. For the computational evaluation study, Oracle Database is used, which provides the widest spectrum of clauses and applicable functions. However, generally, proposed methodology can be generally used in any relational platform. Section 5 deals with the methodology provided as a result summary of the performance evaluation present in section 4.

II. AGGREGATE FUNCTIONS

Database systems offer bunch of embedded procedures and functions, part of the database core, which are generally available. Functions can be called inside the SQL statements, if they pass some prerequisites, like returning only SQL applicable data type, not impacting existing transactions, etc. These functions can be categorized into several groups, like numerical functions, conversion functions, date value management functions, conditional functions and many more. Most of them are single-row functions meaning, they are applied separately for each row. Among from standard functions (available through the *STANDARD* package) and user-defined functions, aggregate and analytic functions are used for the data analysis.

Aggregate functions return a single value for a group of values, rather than on single rows. They take the group, as specified in the *Group by* clause and calculate the output of the data aggregation. In a query containing a *GROUP BY* clause, the elements of the select list can be aggregate functions, *GROUP BY* expressions, constants, or expressions involving one of these. Oracle applies the aggregate functions to each group of rows and returns a single result row for each group. Applicable clauses for the aggregate functions are *Select* and *Having* clause.

Most of the aggregate functions take a single argument (parameter), which can be optionally enhanced by various flags. The following statement calculates number of occurences for each category specified in the *Group by* clause. Asterisk symbol (*) refers to the row existence. In principle, instead of asterisk, any *NOT NULL* attribute can be used, getting the same results.

```
select A, count(*)
 from TAB
   group by A;

select A, count(nvl(B, 1))
 from TAB
  group by A;
```

III. AGGREGATE FUNCTION PARAMETERS

Typically, aggregate functions accept a single argument (parameter), which can be expression, constant, attribute, function-call reference or even asterisk delimiting the processed data set itself. There are many variants, which produce the same results, like *count(*)* and *count(A)*, based on assumption, there are inner joins only (or none) a *A* is mandatory value column. However, the performance can differ. This paper aims to evaluate those performance impacts, mostly reflected by the processing time and costs and create a methodology, how to optimize the performance.

To sharpen that, argument of the aggregate function can be optionally enhanced by the additional keywords and routines, like *Distinct*, *Unique* or *All*. *Distinct* and *Unique* keywords are synonymous meaning, that before the processing, duplicate values are removed and only distinct values are considered as an argument expression. The default option is All, causing that all values are considered, thus, including all duplicates.

Some aggregate functions can work with dynamic windows using windoing_clause, analogous to the analytic

functions. However, impact of this clause is not performance evaluated and will be part of future goals and research.

Complex statistics and data analytics is done using aggregate functions, typical representations are *Min, Max, Sum, Avg, Count, Variance* or *Stddev*.

Aggregate function is calculated for each group and therefore, by their usage, result set cardinality is (can be) reduced.

Furthermore, aggregate functions ignore indefine (*NULL*) values. It consequences in producing *NULL* values as a result of the aggregation, if the processed set is empty. In the following example, the condition Where ensures that the processed set contains no records:

```
Select sum(A)
 From TAB
  Where 1=2;
--> NULL
```

But this does not apply to the aggregation function count, which in that case produces a value of *0*.

```
Select count(*)
 From TAB
   Where 1=2;
```

Thus, even the aggregate function ignores *NULL* values, output of the aggregate function is never empty:

```
Select count(null)
 From TAB
   Where 1=2;
```

The following performance study will therefore focus on 3 areas:

- Impact of various argument references in the aggregate functions and relation to the processing demands.
- Impact of function calls inside the aggregate fucntion pointing to storing results in a pre-mapping repository.
- Impact of analytics used in the *Group by* clause.

A. Argument reference impact

When dealing with the aggregate functions, there are many options, how to reference the arguments. In general, it can consist of constant, numerical expression, function calls or asterisk, making the ability to create complex evaluation. For the aggregate functions, however, it is recommended to make the evaluation as easy as possible, because it commonly relates to the large data set with many rows. To get the number of rows applying the conditions, Count aggregate function, as well as Sum can be used. Therefore, we introduce various options providing the same results by studying the performance impacts. In this part, following aggregate functions are considered:

- **Count(*)**, where asterisk defines the row existence.
- **Count(A)** taking any *NOT NULL* attribute. The research focuses on the general attributes, attributes part of the indexes and primary keys (creating indexes automatically).
- **Count(1)** taking constant numerical value.
- **Count('xxx')** considering constant character string.
- **Count(nvl(B,1))** referencing any attribute by converting it to any real value. Please note, that the stated value "*1*" is rather placeholder than a real values and the format depends on the data type of the associated attribute (*B*).
- **Sum(1)** - we can also process the number of records using the Sum function, where each record will be represented by the value 1.

Please note, that all the mentioned functions will be considered as a aggregate function fashion, since they can also represent analytical oriented approaches.

B. Function calls optimization using the result cache

Functions can be called one time, irrespective of the parameters and returned values. The second option, in case the function is called with the same parameters repeatedly multiple times, is to save the results. And then, in case of calling the given function again, the prepared outputs can be just accessed. Of course, this requires that function to be deterministic, specified explicitly in the function header. To be applied, it cannot affect data, nor to modify the database structure. However, when using analytical-oriented reporting functions that calculate complex values, this requirement is fully met. A function can be pre-fetched in a user-defined structure or embedded memory *Result cache* can be used. The limitation of the user activity is just related to the parameters to be used, generally, resulting in storing parameters and mapping in a pure textual form, making it hard to follow function call using named notation [5].

Result cache substructure of the *Shared Pool* instance memory was firstly introduced in Oracle Database 11g in 2007. It was primarily used for queries, which are executed multiple times. In that case, results were stored in the memory. In parallel, a security system was introduced, which ensures, that if any update on the data was done, particular stored result reference was invalidated. For the manual request the statement should be cached, Select statement itself can be extended by the following hint (/*+*RESULT_CACHE*/*), stated directly after the *Select* keyword. Among that, cache mode can be set on database or session level:

```
Alter {system | session} set RESULT_CACHE_MODE =
     {auto | manual | force};
```

Later, *PL/SQL Result cache* memory structure was introduced, operated with the same session/database parameter. It is applicable only to the functions.

C. Analyzing data with Rollup and Cube considerations

Over the decades, a tremendous increase of reporting, complex analytics and queries could be seen in any area, including environmental data and transport systems. Analytical support is mostly based on the OLAP technology, data warehouses, marts and their variants. Analysis is done across multiple dimensions, however, individual groups should be also evaluated together in a bulk. One of the key concepts in decision support systems is "multi-dimensional analysis" across all necessary combinations of dimensions, like temporal, spatial, product, category, environment impacts, etc.

A. Rollup

Rollup analytical extension enables a Select statement to calculate multiple levels of subtotals across a specified group of dimensions. It also calculates a grand total. It creates subtotals which "roll up" from the most detailed level to a grand total. Rollup takes an ordered list of grouping columns:

Group by Rollup(A,B,C) considers the following groups:

- A, B, C
- A, B
- A
- Grand total

Without the *Rollup* extension functionality, it would be necessary to process the individual groups separately and then union them into a common output. Besides, it should be emphasized that each *Select* statement defined in this way triggers a whole constellation of operations - from creating an execution plan through selecting indexes, obtaining data, and constructing result sets:

```
Select A, B, C, count(*)
 From TAB
   Group by A, B, C
UNION ALL
Select A, B, null, count(*)
 From TAB
  Group by A, B
UNION ALL
Select A, null, null, count(*)
 From TAB
  Group by A
UNION ALL
Select null, null, null, count(*)
 From TAB;
```

In the above example, three dimensions are considered. For the simplicity, aggregate function *Count* is used. Notice, that for dimension number *N*, particular table needs to be evaluated *N+1* times. In case of using complex queries, significant additional processing demands can be identified. Performance evaluation study points to the results.

B. Cube

Cube extension of the *Group by* clause produces cross-dimensional reports. It takes all possible combinations. Thus, if there are N dimensions, totally, 2^N combinations would be produced. Let the dimension set be *(A, B, C),* then the *Cube* produces following categorical reports:

- A, B, C
- A, B
- B, C
- A, C
- A
- B
- C
- Grand total

This, in case of refusing *Cube* extension, eight statements ($2^3=8$ statements, where *3* expresses number of dimensions) would be necessary to be issued. However, by adding one extra dimension, there would be *16* statements. To sharpen that, if *10* dimensions would be present, *1024* statements would be necessary to be launched.

Tab. 1 shows the correlation between the number of dimensions, number of referenced statements using emulation of the *Rollup* and *Cubes*, as well as reduction in percentage.

Graphical representation of the reduction factor for *Rollup* and *Cube* is depicted in Fig. 3.

TABLE I. REDUCTION FACTOR REFERRING TO THE NUMBER OF DIMENSIONS

Number of processed dimensions	Data structure access emulating Rollup	Reduction of the data access, if Rollup is used (in percentage)	Data structure access emulating Cube	Reduction of the data access, if Cube is used (in percentage)
1	2	50	2	50
2	3	66.6	4	75
3	4	75	8	87.5
4	5	80	16	93.8
5	6	83.3	32	96.9
10	11	90.9	1 024	99.9
20	21	95.2	1 048 576	99.9

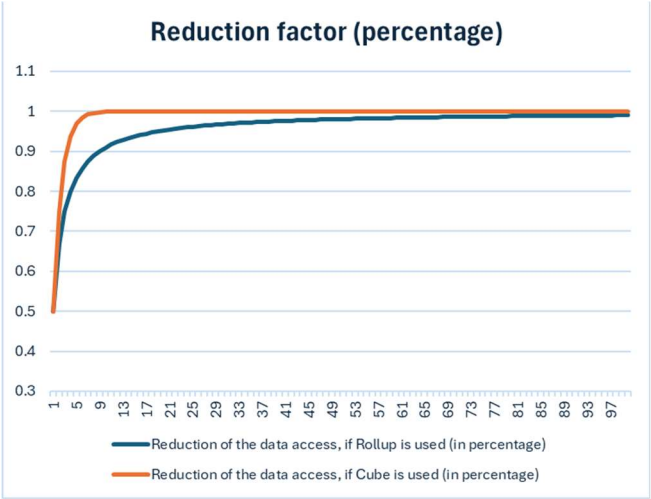

Fig. 3. Reduction factor by using Rollup and Cube extensions

IV. PERFORMANCE EVALUATION STUDY

For the performance evaluation study, real data set of flight monitoring was used, consisting of spatio-temporal model holding airplane locations and *Flight Information Regions (FIR)* in aviation assignment. It took planned and real routes monitored during the whole operation – from flight preparing, taxi, departure, flight itself, up to landing and parking. Temporal attributes of the airplane assignment to the particular *FIR* referred to the entry and exit time. The data set consisted of *234 740* records in the *European region* during *2017 – 2020*.

Environment for the evaluation study was defined by the server with the following parameters:

- *Operating system*: Windows 11 Pro, 22H2
- *Processor*: AMD Ryzen 5 PRO 5650U with Radeon Graphics, 2.30 GHz
- *Memory*: 2x 32 GB DDR-4, 3200MHz, CL20
- *Disc storage*: 2 TB, NVMe, read/write 3500 MB/s
- Oracle Database 23ai Free Release 23.0.0.0.0 – Production Version 23.4.0.24.05

There are many reasons for selecting Oracle Database for the evaluation. First, it is the most complex system, which provides robust performance for the analytical queries. Secondly, during the last years they brought significant changes and new functionality, mainly in Oracle 21c and Oracle 23ai versions. Finally, Oracle serves as an associate partner of the *EverGreen project*. However, although the evaluation study is performed on Oracle technologies and

platforms, the research results are generally applicable to any type of database system and analytical-reporting repository.

A. *Impact of aggregate function parameter type*

The first experiment emphasizes impact various parameters of the aggregate function and its impact of the overall performance. the aim was to collect number of rows for the individual groups. All the defined functions provide the same results, however, as evident from Tab. 3, processing time demands are not the same. In the query, all rows are processed and categorized, taking the whole data set consisting of *234 740* rows.

At first glance, the results may seem strange. The best results were achieved in the case of using a function, which requires function processing, and a context switch between SQL and PL/SQL environments. Compared to taking only static numerical value (*Count(1)*), it achieves a saving of more than 1 second, which reflects more than *7.5%.* The point lies in the processed value itself. The structure of the memory Buffer cache is block-oriented, so a numerical value, regardless of its size, requires the entire block. A *NULL* value has no special memory requirements.

Some can feel that the term asterisk expresses the whole meaning and thus all the attributes. However, as we can also see from the results, that this is not the case, it only expresses the existence of the record. So those results are also very good, compared to the numeric *NOT NULL* attribute (*A*), reaching more than *1.6* seconds (*10.48%*).

The difference between *Sum(1)* and *Count(1)* is minimal, expressing less than *2.5%.*

TABLE II. PERFORMANCE – PARAMETERS OF THE AGGREGATE FUNCTIONS

Aggregate function	Processing time [ss.ff]
Count(*)	**13.993**
Count(A)	**15.631**
Count(1)	**14.074**
Sum(1)	**14.423**
Count(nvl(null, 1)	**13.012**
Count(nvl(B, 1)	**15.896**

B. *Impact of Result cache for calling functions*

The second experiment points to the *Result cache*, limiting the necessity to call the function multiple times. Instead, results of the fucntions are stored in the memory as a mapper between input parameters and provided result. In both types, user defined function was used, attempting to replace an undefined value by any real value. In case of using *Result cache* defined in the function header, total demands were *16.81*1 seconds. Thus the impact *Result cache* management was *1.272* seconds, which incorporates storing function results, but also searching inside the structure using a function call hash reference, From the performance point of view, the total saving was *7.03%.*

TABLE III. PERFORMANCE – IMPACT OF USING RESULT CACHE

Aggregate function	Processing time [ss.ff]
User defined function	**18.083**
User defined function + Result cache	**16.811**

In this case it is important to mention that in our case the function was marked explicitly to cache the values. If automatic selection were selected, the selection of function features would depend on how often the functions are called compared to others in the pool. The results are shown in Tab. 3.

C. *Impact of using Rollup and Cube extensions*

The last experiment was based on the *Group by* clause extensions, which are preferably used in the data analytics allowing to create subtotals and various group splitting. By using *Rollup* and *Cube*, it is possible to dig deeper into data by analyzing dimensions correlations and overall impacts, organized in data warehouses and marts. The important property is, that the data set is scanned only once, the groups and dimension reflections are then processed and calculated dynamically. The last column of Tab. 4 shows, that even multiple *Group by* sections are used (delimited by using *Rollup* and *Cube*), number of processed rows is still the same. Futhermore, using *Rollup* or *Cube* extensions does not bring significant additional demands and processing time requests. Precisely, compared to the original *Group by* clause (reference 100%), *Rollup* extension requires additional *2.385* seconds (reflecting additional *6.58%*). *Cube* extension takes additional *2.472* seconds (reflecting additional *6.80%*). However, results for additional groups and grand totals are calculated. Tab. 4 shows the results for the *3* dimension core.

TABLE IV. PERFORMANCE - GROUP BY CLAUSE EXTENSIONS

Aggregate function	Processing time [ss.ff]	Number of processed rows by aggregate function
Group by	**33.878**	**234 740**
Rollup	**36.263**	
Cube	**36.350**	

V. METHODOLOGY

The primary goal of this paper is to provide a methodology, how to manage and treat aggregate functions. Number of the data, as well as the model complexity become more interconnected and hidden relationships should be identified. Data analytics is characterized by aggregating data across the entire spectrum with the possibility of recording changes over time. In this paper, various temporal architectures are referenced, forming the input data layer for the data warehouses and marts. In order to make a qualified decision, it is necessary to obtain reliable data that can be used to support this decision. The evolution and data changes are commonly aggregated to point to the significant aspects and features. One of the core function categories used in data analysis is just aggregate functions. This paper discusses three aspects of them aiming to create a methodology providing reliable outputs supported by the performance. The first category deals with the parameters of the aggregate functions. Often, it is necessary to get the cardinality of the groups operated by *Sum* or *Count*. Based on the performance evaluation study, it is evident that the same results can be obtained by various functions. The critical factor is just the data type and data block mapping. Besides, the conversion functions are inevitable parts. Namely, constants of the parameters provide better performance, compared to attribute references, even if the given value is already loaded in the memory, since it is needed for previous processing. So the loading process is not critical, rather the reference is important. There is no strong difference between *Sum* and *Count* aggregate functions, both provide almost the same performance, however, the most important aspect is just

reflected by the parameters. Thus, the best option from the performance point of view is to reference constant, even provided as a function result.

The second evaluated stream is associated with the function calls transforming data for the aggregations. In this case, storing results in the *Result cache* memory structure can be useful, reducing the processing time demands by more than 7%. However, the overall improvement strongly depends on the function complexity. In general, the more difficult the processing of the function is, the more significant the importance of pre-storing the results is identified. Sure, by assuming, that the function is deterministic. Here it is important to note that although in our case it was a simple user defined conversion function, we achieved an improvement of more than *7%*. If we used the functionality (*PRAGMA UDF* - User defined function) [6] to reduce the context switch between SQL and PL/SQL environments, the saving would be more than *10%*.

The last treated category relates to the *Rollup* and *Cube* extensions. They are defined in the *Group by* section of the *Select* statement by producing group subtotals. As a consequence, the order of attributes specified there is important, defining the group formulation to be treated. When using such extensions, the defined data set is processed and evaluated only once, even multiple group categories are defined, making the clause extensions very effective, compared to the conventional method of multiple result sets integration (*UNION ALL*), whereas for each statement, the defined data set must be scanned separately.

VI. Conclusions

Data analysis is an inseparable part of the daily life. We can clearly feel it in traffic, when we need to wait in queues and traffic jams, so we could find a better way. Or it is strongly visible in the environment, in which we perceive a significant change in climate and related factors. It is necessary to take historical data, identify patters related to the current situations by aiming to predict the future states and make proper reactions and make qualified decisions. Data are becoming more and more complex with many hidden correlations and relationships. By analyzing data, these dependencies and features can be identified.

This paper deals with the data analysis by pointing to the aggregate functions. It discusses impact of parameters, function calls and *Group by* clause extensions on the processing time. Since the data number is rising, proper performance of the executed statements is critical. When dealing with the data analysis, the problem is even deeper. Therefore, the methodology of aggregating is proposed and handled. As evident from the computational evaluation study, parameters and block granularity of the data are important and primarily impact the performance. Secondly, if there is necessity to reflect function result inside the aggregate function, caching the results can get measurable improvement. Finally, *Rollup* and *Cube* extensions are discussed, allowing to use multiple groups in one query. Their main purposes are to calculate subtotals and grand total, however, it is operated dynamically by scanning the input data set only once.

During the future research, we will point to defining our own structure for storing function results, directly assigned to the query. It is assumed, that a local memory repository can provide better performance, since the searching in the structure will be much faster than a common shared repository. Besides, we will combine functions using *PRAGMA UDF*. It allows to compile user defined function to be primarily used in SQL. As a consequence, impact of context switches between SQL and procedural language environments is limited. Finally, impact of data types part of the aggregate function parameters will be researched – numerical vs. textual values with fixed or variable sizes.

Acknowledgment

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

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

References

[1] M. H. Bohlen, J. Gamper and C. S. Jensen, "How Would You Like to Aggregate Your Temporal Data?," *Thirteenth International Symposium on Temporal Representation and Reasoning (TIME'06)*, Budapest, Hungary, 2006, pp. 121-136, doi: 10.1109/TIME.2006.17.

[2] X. Gao, X. Zhao and X. Tian, "The Development of Evaluation System for Ocean Environment Monitoring Data Analysis," *2015 2nd International Conference on Information Science and Control Engineering*, Shanghai, China, 2015, pp. 970-972, doi: 10.1109/ICISCE.2015.219.

[3] S. Gummedelli, S. Tiruvayipati and S. Vemula, "A Work-Around Methodology for Non-Executable Aggregate Functions on Encrypted Databases," *2023 10th International Conference on Computing for Sustainable Global Development (INDIACom)*, New Delhi, India, 2023, pp. 1232-1237.

[4] A. A. Hadwer, D. Gillis and D. Rezania, "Big Data Analytics for Higher Education in The Cloud Era," *2019 IEEE 4th International Conference on Big Data Analytics (ICBDA)*, Suzhou, China, 2019, pp. 203-207, doi: 10.1109/ICBDA.2019.8713257.

[5] Michal Kvet, *Developing Robust Date and Time Oriented Applications in Oracle Cloud: A comprehensive guide to efficient date and time management in Oracle Cloud* , Packt Publishing, 2023.

[6] M. Kvet and J. Papan, "The Complexity of the Data Retrieval Process Using the Proposed Index Extension," in *IEEE Access*, vol. 10, pp. 46187-46213, 2022, doi: 10.1109/ACCESS.2022.3170711.

[7] A. Londhe and P. P. Rao, "Platforms for big data analytics: Trend towards hybrid era," *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, Chennai, India, 2017, pp. 3235-3238, doi: 10.1109/ICECDS.2017.8390056.

[8] E. Meskovic, Z. Galic and M. Baranovic, "Managing Moving Objects in Spatio-temporal Data Streams," *2011 IEEE 12th International Conference on Mobile Data Management*, Lulea, Sweden, 2011, pp. 15-18, doi: 10.1109/MDM.2011.44.

[9] E. Mozzafari and A. Seffah, "From Visualization to Visual Mining: Application to Environmental Data," *First International Conference on Advances in Computer-Human Interaction*, Sainte Luce, Martinique, France, 2008, pp. 143-148, doi: 10.1109/ACHI.2008.29.

[10] M. Tayab, W. Zhou, M. Zhao and S. Li, "Big data and public services for environmental monitoring system," *2016 11th International Conference on Computer Science & Education (ICCSE)*, Nagoya, Japan, 2016, pp. 139-143, doi: 10.1109/ICCSE.2016.7581569.