Armed Forces Academy of General Milan Rastislav Štefánik in Liptovský Mikuláš

2023 Communication and Information Technologies Conference Proceedings

KIT 2023

12th International Scientific Conference

organized by

Armed Forces Academy of gen. M. R. Štefánik / Department of Informatics

in co-operation with

The Slovak Electrotechnical Society (Organizing Committee)
IEEE Czechoslovakia Section
CAS/COM/SP Joint Chapter of Czechoslovakia Section of IEEE

Sponsored by

Microsoft, HP, Lynx, Aliter

Editors: Július Baráth, Ľubomír Dedera, Miloš Očkay, Michal Turčaník

Hotel GRANIT – Vysoké Tatry - Slovakia
October 11 - 13, 2023

2023 Communication and Information Technologies (KIT)

Published by the Armed Forces Academy of General Milan Rastislav Štefánik, Liptovský Mikuláš, Slovakia, October 2023

All technical inquiries should be sent to:
Armed Forces Academy of gen. M. R. Štefánik
Department of Informatics
Demänová 393
031 01 Liptovský Mikuláš
Slovakia
Phone: +421 960 422986, +421 960 422735
Fax: +421 960 422269
E-mail: verejnost@aos.sk
URL: www.aos.sk

IEEE Catalog Number CFP23M13-USB (USB)
ISBN 979-8-3503-3838-6 (USB)
IEEE Catalog Number CFP23M13-ART (Online)
ISBN 979-8-3503-3839-3 (Online)

All the articles were reviewed by the members of Technical Program Committee. (Multi-blind peer review)

Content

Centralized Communication Scheduler for LoRa 6
(Vanesa Milonová, Mátyás Neilinger, Ladislav Zemko, Pavel Čičák)

Practical Aspects of Attacks Against Remote MS Windows Corporate Environment 13
(Martin Pavelka, Ján Laštinec)

Heterogeneous Wireless Sensor Networks Enabled Situational Awareness Enhancement for Armed Forces Operating in an Urban Environment 21
(Pawel Kaniewski, Janusz Romanik, Krzysztof Zubel, Edward Golan, Maria D. R-Moreno, Paweł Skokowski, Jan M. Kelner, Krzysztof Malon, Krzysztof Maślanka, Emil Guszczyński, Łukasz Szklarski, RR Venkatesha Prasad)

Improved Visibility of LoRa Networks Using LoRa Performance Evaluation Tool 29
(Kristián Rončkevič, Alexander Valach, Pavel Čičák)

Carrier Sensing of LoRa@FIIT Devices 36
(Michal Greguš, Alexander Valach, Pavel Čičák)

LoRa Industrial Monitoring and Management Network 45
(Dominik Bucko, Alexander Valach, Pavel Čičák, Marcel Baláž, Ondrej Kachman)

Anomaly detection from TLE data 51
(František Dráček, Jiří Šilha, Roman Ďurikovič)

Docker-based Assignment Evaluations in E-learning 56
(Jakub Dubec, Rastislav Bencel, Ján Balažia, Pavel Čičák)

Localization of Modulated Signal Emitters using Doppler-based Method implemented on Single UAV 61
(Rafal Szczepanik, Jan M. Kelner)

Detection of Selected Attacks Based on ANN and Classifiers 66
(Július Baráth, Michal Turčaník)

Dangling Predicates and Function Call Optimization in the Oracle Database 70
(Michal Kvet)

Oracle Application Express as a Tool for Teaching Web Software Development 78
(Ivan Pastierik, Michal Kvet)

Application of Neural Networks in Data Communication Analysis 85
(Bianca Badidová, Michal Turčaník)

Design of Automatic Radiation Detection System as a Part of Radiation Protection in Linear Accelerator Facilities 92
(Martin Dugas, Norbert Ferenčík, Radovan Hudák, Zuzana Naďová, Petra Kolembusová, William Steingartner)

Coherent Direction Finder Model with Quadrature Signal Processing in the HF Band 97
(Miroslav Pacek, Jozef Perďoch, Zdeněk Matoušek)

Adverse Media Screening Portal 103
(Richard Marko, Michal Ries)

Generative Neural Networks as a Tool for Web Applications Penetration Testing 109
(Petr Gallus, Marcel Štěpánek, Tomáš Ráčil, Petr Františ)

A Modified STAP Algorithm for Ground-Based Radar 114
(Jana Ľoncová, Ján Ochodnický)

Improving the Quality of Automated Vehicle Control Systems using Video Compression Technologies for Networks with Unstable Bandwith 120
(Volodymyr Khilenko, Andrii Zinchenko, Marek Galinski, Volodymyr Danylov)

Dangling Predicates and Function Call Optimization in the Oracle Database

Michal Kvet
Department of Informatics, Faculty of Management Science and Informatics
University of Žilina
Žilina, Slovakia
Michal.Kvet@fri.uniza.sk

***Abstract*—Oracle Databases have always been considered the driving engine of information systems performance. Currently, it is not enough to keep only current valid records but also historical, as well as future plans need to be treated, forming a temporal database layer. The input data themselves are treated in various manner, processed, and stored. To ensure performance, database indexes are created to locate the row in the physical storage efficiently by focusing not only on the attribute values but also function results defined in PL/SQL. This paper emphasizes function development in PL/SQL by pointing to the function-based indexing and optimization for the SQL usage reducing content switch. Besides, it deals with the dangling predicates for the CASE command to ensure proper definition and usage methodology.**

Keywords—dangling predicates, temporal databases, CASE, function-based indexing, virtual column, BeeAPEX, EverGreen

I. INTRODUCTION

Database systems, related technologies, and the data layer itself form the critical layers of information processing technology. Currently, we can hardly imagine systems that do not contain a large amount of data, often in heterogeneous formats. Decision-making systems and intelligent information systems require significant trusted data to be handled, processed, stored, and evaluated, forming the additional layer based on reliability, consistency, and security. Relational databases rely on the strictly defined data format delimited by the data model, consisting of the data tables – entities and relationships [4], [7]. They were first introduced in 60ties of the 20th century and are still hugely widespread because of the reliability, data consistency, and integrity, which are supervised by the transactions [8], [15].

The relational transaction ensures the data traverse from one consistent state to another, which is also consistent. The transaction itself is formed by the ACID properties – atomicity, consistency, isolation, and durability, while the integrity is defined by the column, user, relational, entity, and domain property types [11].

It was supported by the Erasmus+ project: Project number: 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.

One factor in the suitability and continuous expansion of relational databases is the correctness of the data done on the integrity rules. The other essential part is the performance in the entire spectrum. Temporal databases were first introduced soon after the first releases of relational databases. The preliminary technology was based on object identifier extension by the validity time frame. Thus, the primary key was formed by the original object identifier and validity borders, expressed by the BD (begin Date of the validity) and ED (end date of the validity). The granularity precision was done for the seconds, and later shifted to the mili and microseconds. Currently, the finest accuracy used in relational technology is one nanosecond. Subsequently, several streams could be identified, either in terms of the temporal spheres by modeling validity, transaction references, or various timestamps characterizing the processing flow and key parts, like insertion, timestamp of the processing start, evaluation timestamp, loading timestamp, transaction approval, etc. Thanks to that, the whole process of the data transformation, up to storing them covered by the approved transaction, can be handled [10].

Among the processed temporal spheres, it is always inevitable to cover the processed granularity properly. Object-level temporal architecture is characterized by the object identifier extension by the temporal elements, allowing to store multiple versions for one object. However, the temporal transaction manager must ensure that no more than one object state is valid at any timepoint. Thus, it allows storing the whole evolution of the object states by placing them in the timeline. The main disadvantage of object-level temporal granularity is the storage efficiency. If the whole state is not changed in case of the update operation, duplicate values are stored because the original value must be copied, forming the new state, even if the original attribute value is not changed. To serve the storage and performance optimization, attribute level granularity was introduced. Such an architecture is based on associating attributes with the temporal spheres (mostly represented by the validity or temporal transaction reference). Thus, the state itself is formed by the projection of individual attributes at a defined timepoint. Attribute-oriented granularity does not provide duplicate tuples and does not suffer from the necessity to store the whole state in case of reaching and update operation. On the other hand, the object image must be formed by sequential processing of each attribute projection [10]. The intersolution is covered by the group-level granularity, which automatically detects the data and creates synchronization groups, which are then treated and referenced as a single unit. Each temporal synchronization group is encapsulated by the validity, as well as rules determining, when to create or remove existing groups to ensure performance and related storage demands. Fig. 1 shows the temporal group on the data model level. Each group can

979-8-3503-3839-3/23/$31.00 ©2023 IEEE

be formed by the individual attributes or existing groups. Vice versa, if any group is marked to be dropped, any group it is part of, is dropped, as well, in a cascading manner [10].

To ensure the global performance of the system, it is important to complexly support the database layer through the storage and retrieval efficiency operated by the database indexes by limiting sequential data block scanning. Temporal databases must be empowered by the indexes, mostly defined by the B+tree or bitmap structures. The B+tree is preferred by the transaction-oriented temporal databases, while it maintains efficiency throughout the scalability options. By using Flower index extensions, also undefined values modeled as NULL can be part of the index [12].

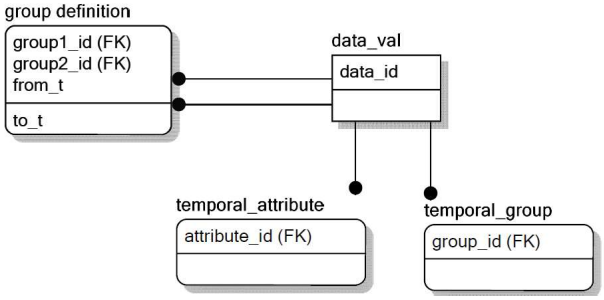

Fig. 1. Temporal group definition data model.

In this paper, we will focus on the function-based indexes in the temporal environment. Namely, the main focus will be on referencing function results, either directly by the index, by the dynamic computation using a virtual column. Another solution used in the analytical databases is a materialized view with auto-refresh [2], [11]. Besides, this paper deals with the CASE command using procedural language by focusing on its extensions and enhancements using Oracle Database version 23c – dangling predicates [18].

The motivation of this paper is to create a methodology for getting the performance of the temporal database dealing with the function-based results. Besides, it analyses new extensions of the CASE statements and expressions to highlight performance impacts.

II. Environment Characteristics

Environment characteristics are associated with the Oracle Database, version 23c, introduced in April 2023. There are several reasons why to use this database system type and architecture. Firstly, it is the most progressive solution by introducing many extensions beyond the ANSI SQL standardization. Secondly, based on the comparison and evaluation, Oracle Database provides the best solutions, respectively is a leading solution. Thirdly, it can be very easily provisioned and maintained in the Oracle Cloud [1], [2]. In the past, the Oracle database system tended to be too complicated to install and set up. It was caused by the ability to complexly configure the environment, parameters, and characteristics [2], [3]. Even though the whole configuration can be done in the cloud environment, as well, it provides an easy provisioning process categorizing workload type to the transaction-oriented, analytical, or JSON type [6]. Fourthly, we have strong skills related to Oracle technologies. Fifthly, it provides the best scalability compared to other database systems and technologies [9]. And finally, this paper is covered by the Erasmus+ projects BeeAPEX [19] and EverGreen [20], in which Oracle is the consortium partner. In BeeApex project, the emphasis is done on the data-driven application development in Oracle Application Express technology, in which the whole processing is referenced by the SQL and PL/SQL, which are then critical from the perspective of the performance. In this project, the emphasis is not only on functionality, but also on processing efficiency and speed. EverGreen project relates to the analytical processing, where the function result referencing and indexing are critical, while the data sets are commonly huge, so the optimization in terms of the data access performance is critical.

Thus, this paper does not focus on the SQL ANSI standard applicable in any database system nor the comparison of the individual, relational database system types. Instead, it focuses only on the Oracle Database and available extensions regarding the CASE expressions, CASE statements, and general function result processing in the temporal models.

For the performance evaluation study, a server with the following parameters was used:

- Processing unit: AMD Ryzen 5 PRO 5650U, 2.30 GHz, Radeon Graphics
- Memory: Kingston, DDR4 type, 2x 32GB, 3200MHz, CL20
- Storage: 2TB, NVMe disc type, PCIe Gen3 x 4, 3500MB/s for read/write operations
- Operating system: Windows Server 2022, x64
- Database system: Oracle Database 23c, release bundle Oracle 23c Free, Developer Release Version 23.2.0.0.0.

The data set used for the performance evaluation study was spatio-temporal oriented, describing airplane positions and Flight Information Region (FIR) assignment. The temporal characteristics were based on the GPS positional data, FIR entry and exit type, and 100 parameters describing the flight. The whole data set consisted of 5 million rows. The CASE was used to identify airplane position and reference to the particular country or region (FIR does not reference the borders of the countries precisely). The structural example of the data source is shown in Fig. 2.

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

Fig. 2. Data source.

III. Referencing Function Results

B+tree index structure is a robust architecture enhancing the performance by optimizing access to the individual tuples, which are block-oriented. Thus, instead of sequential block-by-block scanning to locate the row, an index is to obtain the address of the row in the database layer. ROWID address is the most suitable solution to reference and access the particular row. The B+tree index traversing starts with the root node, through the internal nodes, up to leaf layer, which holds the ROWID references. The whole structure is index key oriented and is always balanced, which can partially limit the loading and update operations. Therefore, post-indexing has been introduced in [12] to exclude the index balancing process from the separate transaction by preserving the reliability of the index by covering all the records. The architecture of the B+tree index is shown in Fig. 3. The key of the B+tree index can be either the attribute itself, a set of attributes, or any

expression resulting from the function call. Besides, the virtual column can also be indexed. The virtual column looks like an ordinary table attribute, but the values are not stored physically in the database. Instead, the content is generated dynamically on demand. The definition is done by the expression to determine the value, optionally enhanced by the GENERATED ALWAYS or VIRTUAL keywords, but they are used just for clarity. The syntax of the virtual column definition is the following [5], [11] :

```
column_name [data_type]
  [GENERATED ALWAYS] as (expression) [VIRTUAL]
```

The data type is also optional. If omitted, the database system determines the type based on the expression results. In the Expression clause, the function result can be referenced. However, the function definition must be deterministic, even stated in the function header explicitly.

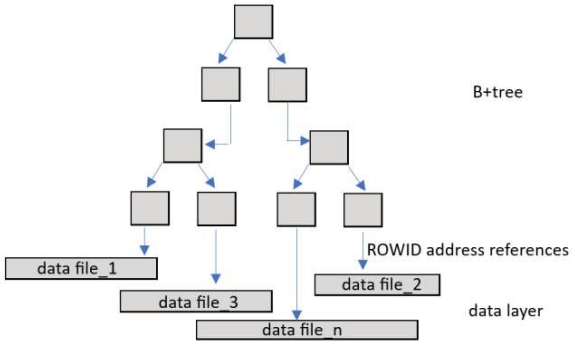

Fig. 3. B+tree index.

Function specification and body are defined by the procedural language (PL/SQL) and then called inside the data manipulation SQL operations (DML statements – Insert, Update, Delete, and Select). While the environment for the SQL and PL/SQL is not the same, content switches must be present, adding additional demands and lowering the performance. Oracle Database, however, allows using a specific routine, optimizing the calls and function result generating by navigating the usage primarily for the SQL statements. PRAGMA UDF (User Defined Function) allows you to navigate the system to optimize the function for the SQL calls instead of the PL/SQL usage.

IV. Enhancing CASE command

The CASE expression and statement have always been part of the conditional processing defined by the procedural language. All database systems have such an option. Although the syntax and keywords can differ, the principles, applicability, and usability are always the same. That principle was adopted by the Oracle Database and was available throughout the versions up to Oracle Database 21c, introduced in December 2020. There were two types of CASE expressions and statements, either defined by the equality mapping or by the conditions. Equality mapping consists of two parts. CASE clause specifies the left side of the equality evaluation, commonly expressed by the attribute, variable or function result. Individual WHEN branches then refer to individual values, typically specified by the constants stated explicitly. The evaluation is done by the equality check in a mathematical manner, consequencing in the inability to treat NULL values, whereas they must be covered by the IS NULL or IS NOT NULL clause, equality check is always considered as NULL resulting in processing in a FALSE branch.

The syntax of the CASE equality mapping is following:

```
CASE expression
  WHEN val_1 then process_1;
  WHEN val_2 then process_2;
  WHEN val_n then process_n;
   [ELSE process_else;]
END CASE;
```

Several WHEN branches can be present, the system looks for the first from the top to apply. Optionally, the ELSE clause can be present to cover all evaluations which do not fit any WHEN branch.

Variables are not initialized, so they hold NULL by default. In the following code snippet, the ELSE clause will be handled because NULL values cannot be mathematically treated, consequencing that the fifth branch will never be applied.

```
declare i integer;
begin
 case i
  when 1 then dbms_output.put_line(1);
  when 2 then dbms_output.put_line(2);
  when 3 then dbms_output.put_line(3);
  when 4 then dbms_output.put_line(4);
  when NULL then dbms_output.put_line('is null');
   else dbms_output.put_line('else');
 end case;
end;
```

The second CASE type takes the empty CASE part. The whole evaluation is done based on the condition stated for each WHEN branch. One branch has one condition, optionally enhanced by the ELSE clause. The evaluation is done from the top. The first branch, which is evaluated as TRUE, is taken. This solution is much more flexible, while it is not limited to managing only conditions based on equality. The syntax is shown in the following code snippet. It can cover any condition, even based on a NULL value, while the processing is treated based on the condition, not just by the equality. The disadvantage of this approach is the function result reference, which can be evaluated multiple times, one for each branch, compared to the first CASE type, in which the value is obtained only once.

```
CASE
  WHEN condition_1 then process_1;
  WHEN condition _2 then process_2;
  WHEN condition _n then process_n;
   [ELSE process_else;]
END CASE;
```

The problem is depicted by the following code snippet. Note, that the referred function can be called up to five times, which brings additional processing time and resource demands:

```
begin
 case
  when get_data=5 then dbms_output.put_line(5);
  when get_data=4 then dbms_output.put_line(4);
  when get_data=3 then dbms_output.put_line(3);
```

```
  when get_data=2 then dbms_output.put_line(2);
  when get_data=1 then dbms_output.put_line(1);
   else dbms_output.put_line('else');
 end case;
end;
```

Generally, the problem can be even deeper. Although it can be partially done by pre-storing function results, it is still an uncommon and non-general solution:

```
declare
 processed_val integer;
begin
 processed_val:=get_data;
 case
  when processed_val=5 then dbms_output.put_line(5);
  when processed_val=4 then dbms_output.put_line(4);
  when processed_val=3 then dbms_output.put_line(3);
  when processed_val=2 then dbms_output.put_line(2);
  when processed_val=1 then dbms_output.put_line(1);
   else dbms_output.put_line('else');
 end case;
end;
```

All the above solutions apply ISO standard – SQL:2003 (ISO03a, ISO03b) spread by the database systems. Oracle Database 23c introduced several CASE expression and statement extensions to make it more flexible [13], [18]. It allows to combine both types and encapsulate both principles and approaches in a single unit. Namely, the CASE clause can consist of the expression reference, while individual WHEN branches can consider multiple values or ranges. Instead of equality mapping and evaluation, dangling predicates can be used. A dangling predicate is an expression which lacks the left operand. Namely, if the mathematical operation is missing, the equality symbol is automatically used. The principles are shown in the following code snippet. The first branch takes the equality value, while the rest are based on the range limits.

```
declare
 output varchar(100);
begin
 output:=case get_data
            when =0 then 'landed'
            when <=10 then 'taking off'
            when <500 then 'flying'
            when <700 then 'landing'
            when <800 then 'taxi'
            when is null then 'unknown'
          end ;
end;
```

Please note that the equality sign can be omitted.

```
output:=case get_data
          when 0 then 'landed'
         end;
```

The NULL value must be treated by the IS NULL keyword manner. Otherwise, it would be treated by the equality sign resulting in the inability to cover such a branch. The stated enhancement allows the user to reference the function call without the necessity to declare a local variable and store the function result there.

A. Multiple conditions in a single WHEN branch

A single WHEN clause can contain multiple checks and conditions done by the dangling predicates, separated by commas. The fourth WHEN takes a list of three values covered by the branch. The seventh WHEN defines two conditions.

```
declare
 output varchar(100);
begin
 output:=case null
            when =0 then 'landed'
            when <=10 then 'taking off'
            when <500 then 'flying'
            when 100, 200, 300 then 'checkpoint'
            when <700 then 'landing'
            when <800 then 'taxi'
            when >800, <0 then 'error'
            when is null then 'unknown'
          end ;
 dbms_output.put_line('value:' || output);
end;
```

B. CASE statement vs. CASE expression

CASE statements and CASE expressions are similar in form and functionality. CASE expression produces a single value, while CASE statement is the execution of the PL/SQL command sequence. There is a processing (assignment) done for each WHEN branch separately. Syntactically, whereas multiple commands can be present in a CASE statement, each of them is terminated by the semicolon (;). Besides, it ends with the END CASE, while the CASE expression is closed by the END clause only.

V. PERFORMANCE STUDY

The computational performance evaluation study can be divided into three categories processed separately. In the first part, data selection is treated and enhanced by various index types. The selection itself is made in the numerical value comparison, character string comparison, and date time value processing. The second evaluated category deals with the conversion functions, either by calling packaged function (package DBMS_STANDARD), part of the database system, own PL/SQL function, user-defined function optimization to SQL environment (using UDF pragma clause), and implicit conversion. The third evaluation stream deals with the CASE statement and expression, extended by the enhancements introduced in Oracle 23c (like dangling predicates).

A. Data selection

This evaluation category takes into account the relational algebra operation selection [16], [17] by limiting the data amount by the Where clause of the Select statement. The selection limited the data set to 1%, 10%, 30%, 50%, and 80%, compared to the whole data set, consisting of 5 million rows. The selection was made based on the function result. There were two indexes (B+ tree and Bitmap), which considered the input value, not the function results themselves. Then, a function-based B+tree was developed. Another solution for the consideration was based on the virtual column, which was then indexed. The advantage of the virtual column is that no additional storage demands are necessary, while the value is calculated on demand and can be memory buffered. However, based on the results, the processing time and global processing demands deeply depend on the data type to be processed. The

results are shown in Tab. I. Please note that the system was forced to use the defined index by the hint part of the Select clause. As evident, whereas only function input values were indexed for the B+tree and Bitmaps, the performance was even smaller than sequential data block scanning (Table Access Full – TAF), because the whole index was necessary to be scanned, followed by the function result calculation, processing and evaluation. Although the row address was directly available, the processing lacks the wider ability to scan the data in parallel [5], [13], [14].

TABLE I. PERFORMANCE - INDEXES PROCESSING FUNCTION RESULTS – INTEGER.

selection INTEGER	TAF	B+tree	Bitmap	Function based B+tree	Virtual column indexing
1%	0.40	0.42	0.54	0.43	0.47
10%	4.21	4.22	4.11	4.75	4.22
30%	12:34	12.44	12.48	12.89	13.20
50%	20:08	21.67	21.38	22.63	22.34
80%	35:93	34.83	34.69	34.92	35.82

While dealing with the character string in the conditions, significantly different results were obtained. The evaluation of the string is much more demanding in all categories. The significant processing time increase can be identified for the virtual index, as well. Compared to the function-based index, the virtual column requires almost double the time. Based on further analysis, it is caused by the different principles in the memory storage – function results are stored in the Result Cache of the Shared Pool, while virtual values are placed in the memory Buffer cache. The character string evaluation is more complicated than numerical or Date values because of the heterogeneity in the structure and dynamic size (varchar type was used). The results are shown in Tab. II.

TABLE II. PERFORMANCE - INDEXES PROCESSING FUNCTION RESULTS – CHARACTER STRING.

selection STRING	TAF	B+tree	Bitmap	Function based B+tree	Virtual column indexing
1%	1.34	1.33	1.33	1.35	2.34
10%	12.95	13.40	13.51	13.22	24.19
30%	40.82	40.97	40.99	41.86	73.99
50%	67.55	70.83	70.53	68.96	121.43
80%	109.32	109.82	109.80	114.00	200.70

The last condition evaluation category emphasizes Date values. The proportions between individual types are similar to the numerical types stated in Tab. I. Function-based results and virtual columns provide almost the same performance. The results are in Tab. III. For the TAF, all the blocks need to be scanned. The increase in the processing time is therefore associated with the result set building in the memory, forcing the system to free up the space there. The server costs are almost the same, irrespective of the number of data in the result set for TAF, B+ tree, and Bitmap. A different situation is related to the function-based results and virtual columns. Although the processing time is almost the same for both types, differing up to 4.76%, the difference in the total server demand ranges from 5.9% up to 8.3%. The reason is that total processing costs refer to various server parameters and consumption details, like memory, I/O, processing time, as well as the number of used background processes.

TABLE III. PERFORMANCE - INDEXES PROCESSING FUNCTION RESULTS – DATE.

selection DATE	TAF	B+tree	Bitmap	Function based B+tree	Virtual column indexing
1%	2.45	2.34	3.01	2.31	2.24
10%	23.78	23.55	25.38	24.03	23.55
30%	25.11	74.00	73.33	73.49	73.40
50%	122.44	123.39	125.69	122.37	121.21
80%	196.97	197.42	198.32	128.44	122.42

B. Referring functions

SQL language can serve function calls if they pass some pre-requirements. Firstly, inside, there can be no Data Definition Language (DDL) command nor transaction end (Commit, Rollback). If called by the Select statement, no data change operation can be present (Insert, Update, Delete). Secondly, all the used routines and data types should be supported by the SQL (e.g., many database systems, as well as Oracle Database releases prior 23c do not support Boolean type). For the evaluation study, conversion methods are evaluated, taking the Date representation transformed to the character string value. The reference solution is the TO_CHAR conversion method, part of the STANDARD package available in the Oracle Database. It takes 100%. Explicit conversion function definition in PL/SQL requires an additional 21 up to 23%. As evident, there is no significant difference between SQL and PL/SQL optimization, and UDF pragma does not provide relevant improvement in the temporal environment. Finally, relying on the implicit conversion brings a significant performance jump by lowering the processing time demands by 46.80%. Thus, from the performance point of view, implicit conversions should be preferred. On the other hand, relying on implicit conversions can bring significant reliability issues. Namely, it refers to the server or session format (delimited by the NLS_DATE_FORMAT parameter), which value is initially inherited from the server parameter. Thus, if the National Language Support (NLS) parameter is changed, a different output format would be produced. Explicit TO_CHAR function or own definition can be prone to the parameter settings.

Graphical representation is shown in Fig. 4.

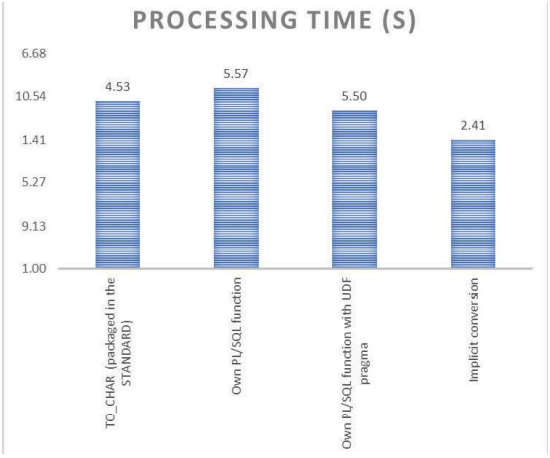

Fig. 4. Processing time – Function definition types.

The results are in Tab. IV.

TABLE IV. PERFORMANCE – FUNCTION DEFINITION TYPES.

SQL function calls	Processing time (s)	Percentage
TO_CHAR (packaged in the STANDARD)	4.53	100.00
Own PL/SQL function	5.57	122.96
Own PL/SQL function with UDF pragma	5.50	121.41
Implicit conversion	2.41	53.20

C. Using CASE statements and expressions

During the performance evaluation, we also dealt with the CASE expression and statement usage by focusing on the dangling predicates introduced in Oracle Database 23c. There were three streams for the evaluation based on the processed data value type – numeric, character string and date value. For each data type, expressions and statements are considered separately by considering the dandling predicates. Each call was done 5 000 000 times. SOL1 is based on the expression dealing with the numeric value. No dangling predicate was used. Get_data is a function which value is requested for each WHEN branch.

```
output:=case
            when get_data=0 then 'landed'
            when get_data<=10 then 'taking off'
            when get_data<500 then 'flying'
            when get_data<700 then 'landing'
            when get_data<800 then 'taxi'
            when get_data is null then 'unknown'
          end ;
```

SOL2 uses a similar solution based on the numeric value representation and consideration. However, the dangling predicate is used. Thanks to that, the particular function Get_data is called only once for the whole CASE expression:

```
output:=case get_data
            when =0 then 'landed'
            when <=10 then 'taking off'
            when <500 then 'flying'
            when <700 then 'landing'
            when <800 then 'taxi'
            when is null then 'unknown'
          end ;
```

Analogous solutions for dealing with the statements are considered in SOL3 and SOL4:

SOL3:

```
case
   when get_data=0 then output:='landed';
   when get_data<=10 then output:='taking off';
   when get_data<500 then output:='flying';
   when get_data<700 then output:='landing';
   when get_data<800 then output:='taxi';
   when get_data is null then output:='unknown';
end case;
```

SOL4:

```
case get_data
   when =0 then output:='landed';
   when <=10 then output:='taking off';
   when <500 then output:='flying';
   when <700 then output:='landing';
   when <800 then output:='taxi';
   when is null then output:='unknown';
end case;
```

The second category for the reference pre-stores the function results in the local variables, which are then evaluated. Therefore, the function is called only once to store it locally.

SOL5 uses an expression with no dangling predicate, while SOL6 deals with the dangling predicate. SOL7 is covered by the statement with no predicate, and SOL8 emphasizes the statement with dangling predicate usage.

SOL5:

```
data:=get_data;
  output:=case
            when data=0 then 'landed'
            when data<=10 then 'taking off'
            when data<500 then 'flying'
            when data<700 then 'landing'
            when data<800 then 'taxi'
            when data is null then 'unknown'
        end ;
```

The results for the numeric value consideration are shown in Tab. V.

Based on the results reached, by using dangling predicates for the function calls, the overall processing time demands are almost the same as pre-storing. Furthermore, even some small improvements can be identified, caused by optimization in PGA (Process Global Area), because the local variables do not need to be declared and managed explicitly. For SOL1 and SOL3, the significant additional demands are caused by repeated function calls for each branch of the CASE expression or statement. In our case, 6 WHEN branches were present. The best solution was obtained by the SOL2 – expression with dangling predicate usage – 0.801 seconds. Compared to SOL1 and SOL3, they would require an additional 192.0% for SOL1 and 137.7% for SOL3. Statement usage is preferred over the expression for all the cases by reducing the demands ranging from 0.6% to 18.6%. A graphical representation of the results is stated in Fig. 5.

TABLE V. PERFORMANCE – CASE IN PL/SQL – INTEGER.

Name NUMERIC	Type Expression (E) Statement (S)	Dangling predicate	Pre-storing function result	Processing time (s)
SOL1	E	✖	✖	2.339
SOL2	E	✓	✖	0.801
SOL3	S	✖	✖	1.904
SOL4	S	✓	✖	0.765
SOL5	E	✖	✓	0.781
SOL6	E	✓	✓	0.807
SOL7	S	✖	✓	0.732
SOL8	S	✓	✓	0.802

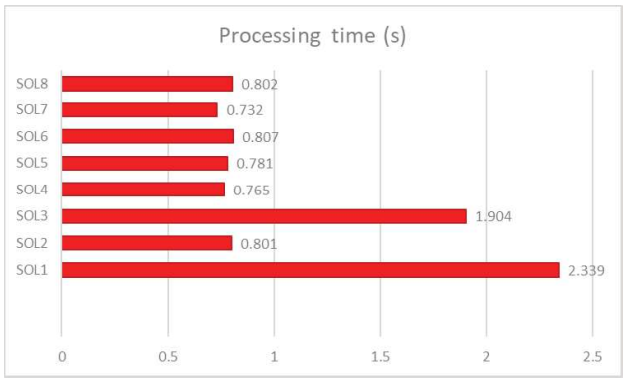

Fig. 5. Processing time – CASE – integer value consideration.

When dealing with the character string as an output of the function, significant additional processing time demands can be identified, compared to the numeric value processing, which is done by the wider range of the applicable values, diacritics, lowercase, uppercase formats, as well as alpha-numeric characters, which can be applied. Tab. VI shows the results. The worst solution was obtained by the solutions with no dangling predicates. The total demands are 3.660 seconds for expression handling and 3.566 seconds for the statement. It is caused by the necessity to recalculate function results multiple times for the same parameters (once for each CASE command branch. Although it can be partially limited by caching the result of the function using the RESULT_CACHE clause [5], [14], it is still necessary to reference the function and find the value in the memory structure of the database instance. Generally, the size of the Result Cache of the Shared Pool is limited, so already calculated value can be removed to serve other workloads. By using a dangling predicate, total processing time demands are lowered up to 1.110 seconds for expressions, which expresses a 69.67% decrease. For the statements, the decrease refers to 70.58%. By comparing SOL 2 and SOL5 solutions, the impact of the pre-storing can be taken into account. Even using a dangling predicate ensures lower processing time demands, while the value is stored internally with no necessity to allocate and reference local variables. Namely, it requires only 1.110 seconds, expressing only 95.16%. Similarly, for the CASE statement, by considering SOL 4 and SOL 7 solutions, total demands express only 72.70%. A more significant decrease is caused by the type and size of the data format – variable size of the character string.

TABLE VI. PERFORMANCE – CASE IN PL/SQL – STRING.

Name STRING	Type Expression (E) Statement (S)	Dangling predicate	Pre-storing function result	Processing time (s)
SOL1	E	✗	✗	3.660
SOL2	E	✓	✗	1.110
SOL3	S	✗	✗	3.566
SOL4	S	✓	✗	1.049
SOL5	E	✗	✓	1.167
SOL6	E	✓	✓	1.205
SOL7	S	✗	✓	1.443
SOL8	S	✓	✓	1.135

The intersolution between the numeric and textual representations is formed by the DATE value processing. The correlations between individual solutions are almost the same, taking 87% of the processing time between Date and already discussed character strings. Although it benefits from the precise storage demands, the structure and formats are influenced by the national language set formats. Thus, if any of those parameters are changed, the already pre-prepared result cache holding transformed data should be invalidated, as well. Thus, the dangling predicate is then more performance effective.

TABLE VII. PERFORMANCE – CASE IN PL/SQL – DATE.

Name DATE	Type Expression (E) Statement (S)	Dangling predicate	Pre-storing function result	Processing time (s)
SOL1	E	✗	✗	3.184
SOL2	E	✓	✗	0.985
SOL3	S	✗	✗	3.104
SOL4	S	✓	✗	0.901
SOL5	E	✗	✓	1.027
SOL6	E	✓	✓	1.048
SOL7	S	✗	✓	1.321
SOL8	S	✓	✓	1.001

VI. CONCLUSIONS

The relational database paradigm is still the widespread concept of storing data in the data model formed by the entities and relationships delimited by integrity constraints. All the characteristics and data suitability are ensured by the transactions shifting the database to a consistent state before the transaction approval. In this paper, Oracle Database is used, which is characterized by robustness, reliability, performance, and scalability. It can be used either on-premise or cloud, in which the supervision of the database, availability and patching is done by the cloud vendor. The treated environment is based on temporal data processing.

This paper aims at the temporal database function management by combining procedural (PL/SQL) environment and world of the SQL language. Several architectures and enhancements were discussed. The focus was done on the content switches between SQL and PL/SQL, which can be reduced by using the PRAGMA UDF clause, navigating the system to compile user defined functions for SQL usage.

Besides, the emphasis was done on the conversion between individual data types. Whereas the core parts are temporally treated, conversion of the Date value to the character string was used. It can be done by various techniques and solutions, either by the function provided in the STANDARD package of the Oracle Database bundle, by explicit or implicit conversions. Although the best performance is obtained by the implicit conversion, such a solution is not robust, whereas it strongly depeknd on the server or session National Language Support (NLS) parameters. Thus, by changing the value of the NLS_DATE_FORMAT, different results would be produced. Then, the undefined value roots are discussed by focusing on the B+tree index extension to cover undefined values, as well as function result indexing optimization.

The second part of the paper deals with the Oracle Database 23c release extension, introduced in April 2023. It provides a bunch of new features and optimization techniques to sharpen the performance. One of the key features studied in this paper relates to the dangling predicate CASE command in the PL/SQL. Therefore, in the computational study, the impact of the CASE statement and CASE expressions is analyzed, by focusing on the transformations and usability. Dangling predicates are characterized by the missing left operand of the expression. It provides significant performance improvements if function results are referred, there, even better than pre-storing function result in a local variable declared in the function, whereas it requires the declaration, assignment, and checking. In the case of using a dangling predicate, the whole management is left to the database system optimizer, which stores internal variables more efficiently within the session memory (PGA). Moreover, the entire declaration is created immediately before the assignment or references themselves. This has the effect of reducing processing demands on the memory space and consumed capacity.

In the future, we will emphasize other key features of the Oracle Database 23c by focusing on the performance impacts and overall methodology. Additionally, we will focus on optimizing the performance of indexes in spatio-temporal databases. We assume that the combination of B+tree and bitmap bucketing indexes covered by the function results in the temporal database environment can bring additional power because the timestamp of the validity start does not need to be the same as the insertion timestamp.

ACKNOWLEDGMENT

It was supported by the Erasmus+ project: Project number: 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.

REFERENCES

[1] A. Abhinivesh and N. Mahajan, “The Cloud DBA-Oracle,” Apress, 2017.

[2] L. Anders, “Cloud computing basics,” Apress, 2021.

[3] T. Cunningham, “Sharing and Generating Privacy-Preserving Spatio-Temporal Data Using Real-World Knowledge,” In 23rd IEEE International Conference on Mobile Data Management, Cyprus, 2022.

[4] R. Greenwald, R. Stackowiak, and J. Stern, “Oracle Essentials: Oracle Database 12c,” O'Reilly Media, 2013.

[5] S. Idreos, S. Manegold, and G. Graefe, “Adaptive indexing in modern database,” In ACM International Conference Proceeding Series, 2012.

[6] M. Jakóbczyk, “Practical Oracle Cloud Infrastructure: Infrastructure as a Service, Autonomous Database, Managed Kubernetes, and Serverless,” Apress, 2020.

[7] J. Janáček and M. Kvet, “Shrinking fence search strategy for p-location problems,” In 2020 IEEE 20th International Symposium on Computational Intelligence and Informatics (CINTI), Hungary, 2020.

[8] D. Kuhn and T. Kyte, “Oracle Database Transactions and Locking Revealed: Building High Performance Through Concurrency,” Apress, 2020.

[9] Y. Kumar, N. Basha, K. Kumar, B. Sharma, and K. Kerekovski, “Oracle High Availability, Disaster Recovery, and Cloud Services: Explore RAC, Data Guard, and Cloud Technology,” Apress, 2019.

[10] M. 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, ISBN: 978-1804611869.

[11] D. Kuhn and T. Kyte, “Expert Oracle Database Architecture: Techniques and Solutions for High Performance and Productivity,” Apress, 2021.

[12] M. Kvet and J. Papán, “The Complexity of the Data Retrieval Process Using the Proposed Index Extension,” IEEE Access, vol. 10, 2022.

[13] J. Lewis, “Cost-Based Oracle Fundamentals,” Apress, 2005.

[14] Z. Liu, Z. Zheng, Y. Hou, and B. Ji, “Towards Optimal Tradeoff Between Data Freshness and Update Cost in Information-update Systems,” In 2022 International Conference on Computer Communications and Networks (ICCCN), USA, 2022.

[15] W. Schreiner, W. Steingartner, V. Novitzká, “A Novel Categorical Approach to Semantics of Relational First-Order Logic,” In Symmetry-Basel, vol. 12, issue 10, MDPI, 2020.

[16] S.Y.W. Su, S.J. Hyun, and H.M. Chen, “Temporal association algebra: a mathematical foundation for processing object-oriented temporal databases,” IEEE Transactions on Knowledge and Data Engineering, vol. 4, issue 3, 1998.

[17] X. Yao, J. Li, Y. Tao, and S. Ji, “Relational Database Query Optimization Strategy Based on Industrial Internet Situation Awareness System,” In 7th International Conference on Computer and Communication Systems (ICCCS), China, 2022.

[18] Case enhancements - https://oracle-base.com/articles/23c/case-statement-and-case-expression-enhancements-23c

[19] Erasmus+ project BeeAPEX - Better Employability for everyone with APEX : https://beeapex.eu/

[20] Erasmus+ project EverGreen dealing with the complex data analytics: https://evergreen.uniza.sk/