Effective Restriction of the Data Manipulation Operations in Oracle Database

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

***Abstract*— Relational theory is built on the foundation of four main operations manipulating data – Insert, Update, Delete and Select providing the data retrieval. However, many times, it is necessary to limit operations changing existing data, forming the concept of the insert-only tables. This paper evaluates performance of individual solutions by creating a methodology of data management and manipulation, which is a critical part of the complex data management and big data term of the intelligent transport systems, pointing to the security option. It takes the reader through the data level protection, privileges, up to immutable tables and retention periods. The evaluation study environment takes Oracle Database 23ai.**

I. Introduction

Relational theory of database systems has a long history. The first database systems were released in 1960s, based on the concept of entities and relationships between them. The whole management of the data was formed on the set of operations, which can be combined by building a robust solution [1].

In the initial phases, the amount of the data was strictly limited, mostly due to the disc and hardware capacity. Simply, the storage media were too expensive and even very corruptible. Over the decades, the reliability, availability and capacity problems disappeared, allowing to store significantly larger ranges of data. Original conventional concept [2] [3] of storing only current valid data was gradually enhanced by the temporal theory [4] [5], allowing to store data over the time, to focus on the data changes on the database, table or attribute level. Temporal theory is still gradually expanding by pointing on the architecture, precision levels and storage principles [6]. One way or another, temporal data management is currently an inseparable part of the data processing. Data processing possibilities resulted in building, developing and maintaining concepts of big data [7] and analytics. Even these days, the need for data analysis is more significant, focusing on the optimizing processes, save natural resources, limiting environmental impacts, etc. Proper decision-making is based on the reliable data, characterized by the timeline evolution reference [8]. Environmental data analytics [9] is one key part of the data management and handling and it is necessary to spread it to daily life, to cover environmental concepts in all daily activities.

A typical example for the big data concept, sensor-based network and complex data layer for the analytics in a temporal manner, is transportation at all levels, from transport of people, goods, supplies, couriers, to air transport [10]. A bunch of data in various structures, precisions and reliability levels is collected, then evaluated and stored, to reflect the current situation, allowing to find optimal routes, evaluating threats, up to identification of accident sections that should be under greater police supervision [8] [10] [11]. In that environment, it is critically important to ensure data consistency [2] [12]. Simply, to ensure, that existing collected data are durable and cannot be changed, making additional security levels. Naturally, data corrections can be done, but only in a temporal manner forming transaction change layer in a bi-temporal system.

This paper focuses on the techniques to ensure the existing data cannot be changed by forming insert-only tables. Such a requirement can be ensured by various techniques and means, from the application level on the one side, through the data layer operated by the triggers, up to privilege definition and accessing another schema. A new concept is defined by marking a table as immutable. This concept was introduced by Oracle Database 23ai. Therefore, Oracle Database technology is used for the evaluation, however, in the future study, emulating that in other database systems will be inspected.

This paper summarizes individual approaches and creates a methodology for creating and accessing data by restricting any update operation.

The structure of the paper is organized as follows: Section 2 deals with the existing solutions by emphasizing triggers, different schema and privilege management. Section 3 introduces immutable tables, enhanced by the retention policy period. Performance evaluation study is present in section 4.

II. State of the Art

To make the data complex, secure and reliable, it is inevitable to monitor and supervise the process of obtaining the data, but mostly to ensure the existing data cannot be changed, deleted or tampered with in any way. Such a requirement arose shortly after the introduction of the first database systems and was addressed in the form of triggers.

Triggers are a piece of PL/SQL code, that are stored in the database and implicitly fired (run), when the associated operation occurs. Traditionally, triggers are fired for the Insert, Update and Delete operations. From Oracle Database 8i, it is possible to create *Data Definition Language* (*DDL*) triggers for system and other data events on database (like servererror, startup, shutdown) or schema level (e.g. logon, logoff). Among PL/SQL, Oracle can also run Java procedure as a body of trigger [1] [2].

A. Designing triggers

Triggers are primarily intended to perform a related action, when a specific operation is performed, to check or set the particular values. In this case, however, the task is opposite – to stop the operation, itself. In Oracle Database, trigger can be fired either before or after the operation itself. In [5] [13], there is a discussion about the definition and performance impacts. For these purposes, *before trigger* type is more suitable, since the *rollback* command for the operation is limited. Instead, the intended operation is immediately stopped.

B. Creating triggers

Triggers are created using the *Create trigger* command. The header of the definition consists of the firing event, positional timing and other clauses. For many database systems, a trigger can be associated with only one operation forcing the developer to call a stored procedure, if multiple operations need to be associated with a common code snippet [1]. Oracle Database is more progressive and allows to use one trigger for multiple operations, even various conditions can enhance the definition in the *When* clause [1]. Thus, Oracle Database provides more powerful and easier definition. The syntax of the trigger is stated in the following code block:

```
CREATE [OR REPLACE] TRIGGER [schema.]trigger_name
  { {BEFORE | AFTER}
    {INSERT | DELETE | UPDATE
            [OF column_name1 [column2 [, ... ]]]}
     OR {DELETE | INSERT | UPDATE
            [OF column_name1 [column2 [, ... ]]]}
     [...]
[FOR EACH ROW]
  [WHEN (condition)]
trigger_body
```

Based on [14] [15], if the trigger body is complex – consisting of more than 60 lines of PL/SQL code, it is recommended to encapsulate it as a stored procedure and just reference it inside the trigger body. From the definition and demands point of view, trigger size cannot exceed 32K [16].

C. Firing triggers

Trigger is fired once the associated operation is to be done. A trigger is defined for one table only but can be spread for multiple operations. In case of emulating insert-only tables, body of the trigger can consist of only one command invoking an exception, which is then propagated to the calling environment resulting in stopping the operation itself, to which the trigger is associated. It can work well; however, it has significant performance limitations. First, trigger body is a PL/SQL script and associated operation is in SQL. Whereas the environments are not the same, *context switches* are required, consuming additional resources, CPU and memory. Secondly, there can be additional logic for the operations itself, so multiple triggers for the particular operation can be defined. Generally, triggers are fired in a random order [16], so there can be many computations and evaluation logic done before the operation itself is stopped. Finally, from the application perspective, raised exception must be processed and presented to the user in an appropriate format. Although it may seem that the impacts are tiny, it is necessary to point to the complex data structures, large amount of data to be handled. When dealing with the temporal data processing in a dynamic transport system environment, overall performance impacts can be significant. Consequently, it may require additional CPU and memory capacity. In a cloud environment, it can be done easily, the parameters can be enhanced by one-click, but it takes additional costs. In case of using on-premises, upgrades can cause an avalanche of changes throughout the whole architecture [15] [12].

D. Logging trigger manipulated data

Another related problem arises from the requirement to log changes, even attempts to do that. That request primarily arises from the security sphere, however, reflecting insert-only tables, it can be demanding. To make the system reliable, logging must be done in a separate transaction, otherwise the logged data would be rollbacked as a result of operation refuse [2] [17]. Fig. 1 shows the problem. From the master transaction, new autonomous transaction must be created, handling the logged data. This logging aspect definition, however, requires new method definition and storing it in a system.

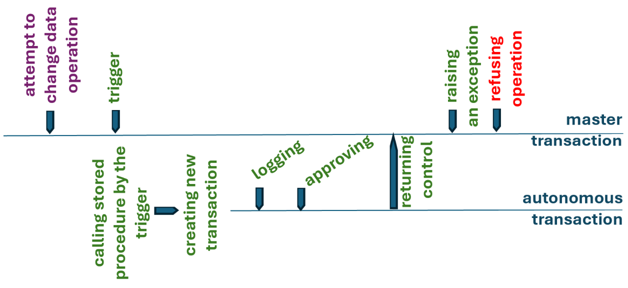


Fig. 1. Data flow & logging secured by the trigger

The message provided to the user depends on the developer definition by invoking *RAISE_APPLICATION_ERROR* procedure. Example of the solution is stated in the following code snippet:

```
create table dv(id integer);

create table logdv(id integer);

create or replace procedure proc_dv(id integer)
is
 pragma autonomous_transaction;
begin
 insert into logdv values(id);
 commit;
end;
/
create or replace trigger trig_dv
before insert on dv
 for each row
begin
   proc_dv(:old.id);
   raise_application_error(-20000,
                        'Insert-only table');
end;
/
```

Please note, that it is infeasible to embed the inner transaction directly to the trigger by raising an exception – *PLS-00710 – Pragma AUTONOMOUS_TRANSACTION cannot be specified here*:

```
create or replace trigger trig_dv
before insert on dv
 for each row
begin
 declare
  PRAGMA AUTONOMOUS_TRANSACTION;
   begin
    insert into logdv values(:old.id);
    commit;
   end;

   raise_application_error(-20000,
                           'Insert-only table');
end;
/
```

E. *Defining access privileges*

One of the common ways, how to limit any operation, can be performed by setting appropriate rights to the data. The database is owned by a user other than the one accessing the application. From the security point of view, the solution is suitable because it separates the database and application layers, not only at the level of the system as such, but also at the level of access and data manipulation. On the other hand, there is the fundamental question of applicability. The fact that a specific user does not have the right to a specific operation does not exactly mean that the operation as such cannot be performed, e.g. by the owner of the database. It is therefore necessary to ensure the access consistency and rights at the level of all users. Moreover, in such a system it is rather difficult to distinguish whether a given user in his role just does not have the right to a given operation or it should be strictly prohibited across the whole ecosystem. In the case of a multi-user environment with a large number of roles, this can be a significant problem. Consequently, it would be necessary to maintain a separate system for managing privileges [6] [18] [19].

Fig. 2 shows the principles for the multi-user environment by using dynamic role (right) mapping. There is an owner of the database and additional layer managing privileges of the ordinary users connecting to the system. Depicted *dynamic right mapping* module is temporal, and rights can evolve over time.

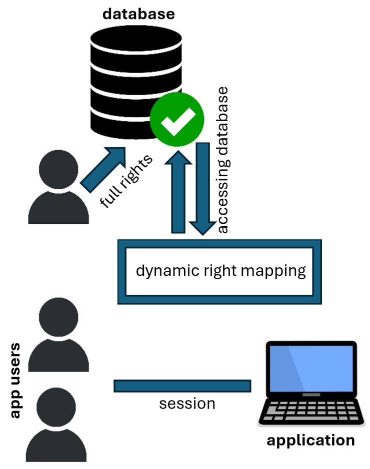


Fig. 2. Dynamic right mapping

F. *Securing operations by the application level*

The last discussed option is based on shifting the whole rule management to the application layer. In that case, application itself is responsible for providing any security aspects and access privileges. On the database layer, everything is allowed. Although this concept partially covers the requirement to limit change operations, any additional interoperability on the data layer can be comprised. Furthermore, it cannot be said that the table is read-only or is flagged as insert-only, since all these operations are still generally available through the database layer.

III. IMMUTABLE TABLE DEFINITION

A. *Immutable tables*

Immutable table definition was introduced in Oracle Database 23ai [19] [20] and back-propagated into Oracle Database 19c and 21c [20] [21]. Immutable tables are insert-only tables in which existing data cannot be changed. Thus, Update statements are prohibited from the definition. Deleting rows is also generally restricted, however there can be a defined retention period – rows can be deleted only after the defined number of days elapsed between the Insert operation and Delete attempt.

Limitation of the immutable table perspective relates to the prohibition of the *DDL*, as well. So, the table structure and layout cannot be later changed. However, *constraint* and *index* management (adding or removing) is still available.

A different limitation is covered by the timing. Namely, the minimum number of days you can specify for the *retention period* is 16 days. Besides, there is a mandatory option specifying the retention of the table itself, defined by the *no drop* clause. The minimum number of days is 0. When defining retention period for the table persistence, be aware, that there is no option to drop table during its retention period other than dropping the whole schema.

Any attempt to update or delete existing rows during the retention period ends by raising an exception *ORA-05715 – Operation not allowed on the blockchain or immutable table.*

1) *Retention period*

By defining the immutable table, it is necessary to specify the retention policy – periods for dropping the table as a whole, as well as the retention for the change operations

Drop table condition:

```
no drop [ until <n> days idle ]
```

Change operation policy:

```
no delete ( [ locked ] |
   ( until <m> days after insert [ locked ] ) )
```

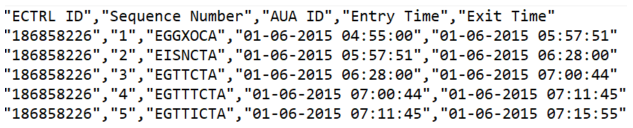
2) *Syntax*

Syntactical definition of the immutable table refers to the *immutable* keyword specified in the table header, followed by the table structure, encapsulated finally by the retention period. For the evaluation study, flights are monitored. Those data cannot be later changed due to the security and reliability reasons. Thanks to this, flight efficiency can be evaluated, which cannot be compromised later.

Example of the definition for the air transport system monitoring is stated below. It consists of flight identifier (*ECTRL_ID*) and sequential number reference, starting from 1 expressing parking at the departure airport, followed by the taxi, takeoff, flying, up to landing, parking. This couple forms the composite primary key. Besides, the table consists of the flight information region reference (FIR) denoted by the *FIR_ID* attribute and temporal assignment – *entry_time* and *exit_time*.

```
Create immutable table FIRmonTAB
  (ECTRL_ID integer,
   sequence_number integer
            check(sequence_number > 0),
   FIR_ID integer,
   entry_time date not null,
   exit_time date,
  primary key(ECTRL_ID, sequence_number))
 no drop until 0 days idle
 no delete until 30 days after insert;
```

Fig. 3 shows an example of the data stored in the *FIRmonTAB* table:



```
"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. 3. Data source structure

Flight information region borders are not strict and evolve over time [8]. Fig. 4 shows the assignment. Please note that FIRs do not reflect physical borders for the countries.

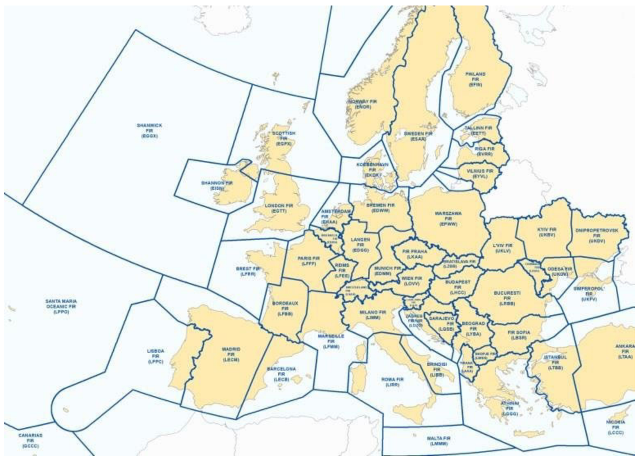


Fig. 4. Flight information regions (FIRs) borders

B. External tables

A specific way limiting change operations can be represented by the external tables, which act as flat files outside the database. The database itself has access to them using *Oracle directory* as a mapping object between the database system and storage repository. From the database layer point of view, external tables are read-only, however, it cannot be ensured, that particular file is inaccessible from the external source. Thus, external tables are not suitable and robust for ensuring insert-only data in a general format.

External table definition consists of the attribute list with associated data types followed by the *organization external* clause [15] [12] [21] [22] [23].

IV. PERFORMANCE EVALUATION STUDY

The environment for performance evaluation study was characterized by the computer 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

Spatio-temporal data set was used, consisting of three subparts:

- *Monitoring flights* - assigning airplanes to the FIRs delimited by the entry and exit time,
- *Planned route* of the planes,
- *Real route* reflecting current circumstances, like weather – wind, storms, restricted areas, etc.

The whole data set consisted of 10 000 flights. Generally, each flight was delimited by 1 000 rows in planned and real route, on average.

The computational evaluation study consists of two experiments:

Experiment 1 deals with the triggers and impact of logging. It takes the data to be updated by storing original primary key values. Two approaches were considered – direct insert into log table embedded directly in the code (*SOL11*) and trigger event storing the data to be handled in the log table (*SOL12*). It is worth repeating that logging data through the trigger requires extracting insert operation into separate autonomous transaction and thus, excluding the code into a separate stored procedure. The evaluation is based on the processing time using the ss.ff format.

Experiment 2 emphasizes limiting change operations using trigger or immutable table. There are three approaches to be considered. The first solution uses application-level limit, so on the database layer, there is no limitation (*SOL21*). That solution can be considered as a reference, since there is no additional impact (on the database management layer). *SOL22* uses trigger with no extra logging, only exception is raised to limit the update or delete operation. Finally, *SOL23* uses immutable table definition – insert-only table is defined on the data model architecture level. Processing time demands (ss.ff) are treated during this experiment, as well.

A. Experiment 1 – Results

Table 1 shows the results of the Experiment 1 dealing with the logging. If there is a request to log data to be handled during the Update operation attempt, direct Insert statement embedded in the code requires 4.435 seconds. If the management of the logging is extracted into trigger, Insert statement into the log table itself is part of the additional procedure invoking autonomous transaction. As evident from the results, it requires 15.124 seconds, which reflects the change of 10.689 seconds. The processing time demands are risen of more than 240%. The reason is based on two facts. Additional requirements and costs are caused by invoking autonomous transaction and context switches. In this case, trigger itself is a PL/SQL block, so the processor must switch between SQL and PL/SQL environment. Additionally, nested stored procedure also requires PL/SQL environment. The automatic log management operated by the trigger is secured and always work independently, but it comes at the cost of increased cost and processing time. The task was to update 10 000 rows in total, reflected as one per thousand of the total amount of data.

TABLE I. EXPERIMENT 1 – PROCESSING TIME RESULTS

	Logging	
	Direct insert into log table	**Trigger** invoking autonomous transaction
	SOL11	**SOL12**
Processing time demands (ss.ff)	04.435	15.124

The graphical representation of the Experiment 1 is shown in Fig. 5.

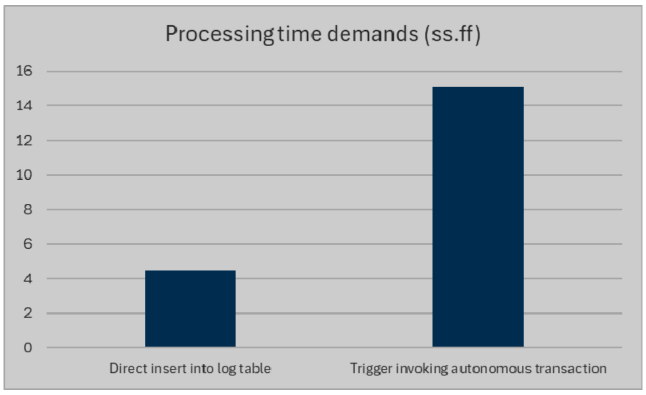


Fig. 5. Experiment 1 - results

B. Experiment 2 – Results

The second part of the study (Experiment 2) comprises impact of insert-only table guarantee. Securing the data table content can be done in various ways. In this study, the focus is done on the trigger, which can raise an exception (application error), if there is attempt to change the content of the table. Trigger can be done on the *row-level*, fired for each accessed row or once for the operation itself – in that case the whole manipulated data are processed and evaluated as a bulk. *SOL21* is a referential solution, in which the Update and Delete operations are unlimited and not restricted in any way, based on the assumption, that the application level secures, that the data are not manipulated or compromised. *SOL22* is based on the trigger definition, version *SOL22a defines row-level trigger* (defined by the *For each row* clause in the trigger definition), while *SOL22b* deals with the *statement-level* trigger. Both triggers are fired before the operation itself. For the statement trigger, the whole data set to be handled is pre-prepared, all data blocks to be changed are created as a single unit. Better solution is provided for the row-level trigger, since if any row causes an exception, the whole processing ends and other data are not processed, not access at all. As evident, it can improve the performance and lowering processing costs (Tab. 2). The last evaluated solution is defined by the introduced *immutable table* (*SOL23*) declaring the aspect of insert-only table on the definition level. Reflecting the performance, the best solution is provided by immutable table (*SOL23*), because no context switch is present. Simply, any attempt of changing the row is immediately refused from the definition. Statement-level trigger switches the environment (SQL -> PL/SQL) once for calling trigger and once for the returning to SQL. Compared to the row-level trigger, in which context switch must be present for each row to be changed, it could be assumed, that an increase in the number of context switches has a natural consequence in an increase in costs and thus also processing time. But it's not like that. The reason is, that there is no necessity to build whole row-set in advance. So if any row causes an exception, processing automatically ends. And it is precisely this fact that can be used when creating insert-only table. Only a single row is handled (generally any row from the set of data for which a change is requested).

The results are shown in Table 2, reflecting additional processing time demands. They do not reflect total processing time, but the additional demands are declared. Therefore, for the the reference solution *SOL21*, value zero is present. The evaluation was performed 10 000 times, in one request, the whole flight was attempted to be updated, reflected by 1 000 rows.

TABLE II. EXPERIMENT 2 – PROCESSING TIME RESULTS

	Insert-only table definition			
	No limitation	**Trigger**		**Immutable table**
	SOL21	**SOL22**		**SOL23**
		Row-level	**Statement-level**	
	SOL21	**SOL22a**	**SOL22b**	**SOL23**
Processing time demands (ss.ff)	00.000	2.420	2.653	2.397

Table 2 shows the results.

Graphical representation of the results is present in Fig. 6.

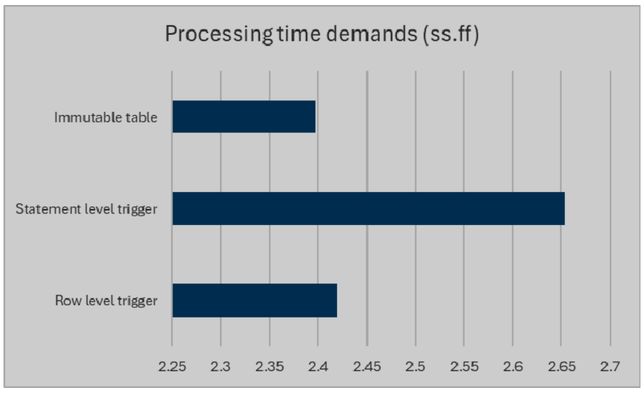


Fig. 6. Experiment 2 – results

The difference between individual trigger types reflects 9.63%, the row-level trigger is more preferred. The best solution is provided by the immutable table, but as declared, the differences are not so huge, primarily caused by the context switch. It reflects less than 1% of additional demands. Therefore, in the future, our focus will be done on the optimization of the triggers.

V. CONCLUSIONS

This paper deals with the Oracle Database by pointing to the performance. For the complex data analytics, there is a common requirement to limit any change on the existing data. In the past, it was primarily ensured by the application layer, however, such a premise is no longer suitable, since the data can be shared and integrated among multiple applications and systems. Therefore, triggers are commonly developed to check the operations, either on the row- or statement-levels. However, their main disadvantage is associated with the calling environment - PL/SQL, which requires context switches from the SQL. In a complex system, it can have significant impact on the performance. This paper uses triggers as a reference solution, enhanced by various extensions and optimizations. It points to the immutable table definition, which limits any change operation directly from the definition. The fundamental difference is based on the calling environment and limits of the context switch, because for an immutable table, a constraint is directly embedded in the data model and used by SQL.

Data analytics is a dynamically developing field of informatics and impacts many areas of the world. It can be found in transport systems, industry, medicine, it strongly points to the environmental data analysis, sustainability and resource saving. Therefore, we assume a huge development and research in the field of the data analytics, data access, structures, as well as indexing, approaches and scalability, mostly in the cloud environment. Namely, there are several streams for the future development and research. Although the immutable table definition seems promising and provides the best performance reflecting the configuration to model insert-only table, it is worth mentioning two facts - immutable tables are exclusively available in Oracle Database 23ai and other systems do not offer them. Therefore, in the next research, we will focus on the trigger management - better results are obtained by the row-level trigger, which can sooner stop the whole operation. In the current situation, order of the rows to be considered and evaluated by the database system is random and depends on the already loaded rows in the instance memory and costs for the loading. Therefore, in the future research, we will evaluate multiple options and impacts on the row order selection, to limit the "random" order used now. Besides, we will focus on implementing analogy of the immutable table to other database systems and approaches making it generally applicable by spreading the concept widely.

Another research stream, we would also like to focus, is related to the retention periods and techniques to incorporate them effectively to the trigger definition, management and handling.

ACKNOWLEDGMENT

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] Kuhn, D. and Kyte, T.: Expert Oracle Database Architecture: Techniques and Solutions for High Performance and Productivity. Apress, 2021.
[2] Kuhn, D. and Kyte, T.: Oracle Database Transactions and Locking Revealed: Building High Performance Through Concurrency, Apress, 2020.
[3] Morris, S.: Resilient Oracle PL/SQL, O´Reolly, 2023.
[4] Kvet, M.: 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
[5] Kvet, M., Papán, J.: The Complexity of the Data Retrieval Process Using the Proposed Index Extension, IEEE Access, vol. 10, 2022.
[6] Castro-Leon E. and Harmon R.: Cloud as a Service. New York: Apress, 2016.
[7] Jakóbczyk, M.: Practical Oracle Cloud Infrastructure: Infrastructure as a Service, Autonomous Database, Managed Kubernetes, and Serverless. New York: Apress, 2020.
[8] Standfuss, T. and Schultz, M.: Performance Assessment of European Air Navigation Service Providers, DASC conference 2018
[9] Erasmus+ project EverGreen dealing with the complex data Analytics: https://evergreen.uniza.sk/
[10] Cunningham, T.: Sharing and Generating Privacy-Preserving Spatio-Temporal Data Using Real-World Knowledge, 23rd IEEE International Conference on Mobile Data Management, Cyprus, 2022.
[11] Steingartner W., Eged, J., Radakovic, D., Novitzka V.: Some innovations of teaching the course on Data structures and algorithms, In 15th International Scientific
[12] Gorelik, A.: The Enterprise Big Data Lake: Delivering the Promise of Big Data and Data Science, O'Reilly Media, 2019.
[13] Malcher, M., Kuhn, D.: Pro Oracle Database 23c Administration: Manage and Safeguard Your Organization's Data, Apress, 2024
[14] Nuijten, A. and Barel A.: Modern Oracle Database Programming: Level Up Your Skill Set to Oracle's Latest and Most Powerful Features in SQL, PL/SQL, and JSON, Apress, 2023
[15] Fotache, M., Munteanu, A., Strîmbei, C., and Hrubaru, I.: Framework for the assessment of data masking performance penalties in SQL database servers. Case Study: Oracle, in IEEE Access, vol. 11, 2023, pp. 18520–18541.
[16] Rosenzweig, B. and Rakhimov, E.: Oracle PL/SQL by Example, Oracle Press, 2023.
[17] Greenwald, R., Stackowiak R., and Stern, J.: Oracle Essentials: Oracle Database 12c, O'Reilly Media, 2013.
[18] Abhinivesh, A., Mahajan, N.: The Cloud DBA-Oracle, Apress, 2017
[19] Anders, L.: Cloud computing basics, Apress, 2021
[20] Immutable table: https://oracle-base.com/articles/23/immutable-table-enhancements-23
[21] Immutable table: https://oracle-base.com/articles/21c/immutable-tables-21c
[22] Hoeren, T. and Kolany-Raiser, B.: Big Data in Context: Legal, Social and Technological Insights, Springer, 2017.
[23] Lee, J., Wei, T. and Mukhiya, S.: Hands-On Big Data Modeling, Packt Publishing, 2018.