Solution Management for Current Temporal Aspect using Tuple Versions

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

***Abstract*— The current temporal paradigm is based on the validity, which delimits each data tuple by the time frame. The uni-temporal solution extends the object identifier with the validity interval inside the primary key. It can be used on various granularities (object, attribute, and group), however, it cannot store individual versions for existing data corrections. This paper references temporal architectures highlights current challenges for dealing with data versions and proposes own solution replacing transaction time frame forming transaction reliability concept. It deals with the referential integrity in the temporal environment and proposes a model for covering references and ensuring the integrity of the processed data using signature hash values, two-level indexes formed by the object and individual states with corrections.**

I. INTRODUCTION

Data stored in the database cover a significant part of the processing. Current database systems are robust and must maintain the whole spectrum of the data with emphasis on the structure, evolution management, and velocity. Security and reliability are the main key spectra to be highlighted, as well.

In the past, data were commonly stored in the files managed by the file systems and accessed sequentially. In the sixties of the 20th century, first database approaches were developed forming a transparent layer between application and data. The most relevant concept is based on relations covered by the mathematical relational algebra [2]. The conventional paradigm as the most complex relational data stream deals with only current valid data [6]. Future valid data are not present in the main structure, as well as historical data. The temporal paradigm was formed in the eighties of the 20th century, by extending object identification using the time definition [6] [7] - oriented temporal approach, which was later improved using various approaches. In that solution, each object is formed by the unlimited number of individual states delimited by the validity frame – begin and end point of the validity [7]. Attribute-oriented approach dealing with column granularity was proposed in 2014 [10]. Each update in such a solution is divided into separate attribute changes and managed separately. In 2017, the hybrid solution was proposed [8] [13]. In that case, the background layer of the attribute-oriented solution is extended by the synchronization processes. Thanks to that, if some data portions are always synchronized, the detector automatically evaluates such a situation and it is maintained similar to the attribute itself. The synchronization group can be composed either by attribute itself or by the group. Internally, each attribute is managed as the group with the set volume.

Limitations of the previously mentioned solutions are just the maintainability if the existing state must be corrected. If the state exists in the system but must be corrected, there is no robust solution. In principle, current systems offer two ways – the existing state is updated with no reflection the history, thus, information about the original state is lost [13]. The second principle is based on correction rejection [4]. Both approaches are, however, limiting. To solve the problem, we propose a reliability extension module for dealing with versions and management of existing data corrections in this paper. It is based on signature hash values management.

II. RELIABILITY ASPECT OF THE TEMPORAL MODEL

Existing solutions do not cover the complexity of existing data changes, the reflection of data corrections. Previously mentioned models are based on validity. They replace the conventional definition by adding validity borders. Thanks to that, each data tuple can be placed on the timeline and ordered [3] [5]. On the other hand, there is no complex module for dealing with data corrections. Imagine, it is not the only problem of replacing existing state due to some mistakes or improper data updates. The problem is much deeper, whereas the environment is fully temporal and deals with future valid data, which can be modified based on changed characteristics and environment. However, there is still a necessity to have relevant information about the upgrade. For such purposes, we propose our own solution, which is data resistant. Thanks to the usage of superstructures, our solution can be connected to any temporal data architecture. Moreover, it finds its place in the conventional paradigm ensuring consistency in distributed and parallel systems. As a consequence, secure, robust, but the most reliable solution is produced. Any data change is stored.

Temporal databases generate complex data environment covering the intelligence of the current information systems, propose data storage and manipulation methods for evaluating, creating prognoses, reactions, decision making, and predictions. Temporal aspects and requirements are constantly improved based on the current requirements and situation. Development characteristics and requirements for temporal models were presented in [3] [5] [6] [10], based on the aspect of usability (easily manageable methods – systems should be able to cover and solve most of the problems automatically

without user intervention necessity) and aspect of performance (there should be no significant slowdown of the system if the conventional approach is replaced by temporal). An aspect of performance also covers the interconnection necessity – existing systems must be able to work without the necessity to rebuild or recompile solutions [7]. It is mostly ensured by the architecture itself (e.g. in attribute oriented granularity, current valid states are stored in the first layer using conventional database principle) or by using views.

In this paper, we extend defined aspects by the requirement of the data structure. It focuses on architecture optimization based on the object data, which should be covered. We propose a categorization of the data tables and attributes themselves: *Conventional* – it is not necessary to monitor data changes over time, the only current valid data image is necessary to be accessible. *Static* – it presents specific subcategory of the conventional model, data do not change their values over time, at all, thus no evolution can be evaluated and presented. In general, particular data updates are prohibited using triggers or access privileges. *Temporal* – data must be monitored and stored over time. With regards to the security and GDPR politics, some historical data must be anonymized or replaced after the defined time interval. It is ensured by the job functionality, in our solution.

The attribute-oriented solution with a hybrid aspect covers all of the previously mentioned categories.

The last added aspect of this paper is based on reliability and security. Data must be managed in a complex manner to hold the tuples with the address of the changes and corrections of existing states. To address the time definition and future valid states, we propose four temporal definitions to solve the conflicts of the data states, whereas each object cannot be defined by more than one valid state anytime. Future valid states, with emphasis on the data architecture, must be automatically transformed into the current if the beginning point of the validity appears. Reflecting the state of the art, our solution uses internal database jobs [1] [2], which ensure, that updates are precise with no delays. Vice versa, although solutions based on operating systems have less system resource demands, they do not ensure, that the transformation is executed in a strictly defined time point. Simply, it cannot be sooner but can be later. In temporal systems with nanosecond granularity, any delay can cause really significant system degradation.

Let have one planned state $S_{planned}$ of the object *O* delimited by the time frame $BD_{planned}$ and $ED_{planned}$ by using closed-open data interval representation (transformations between individual interval types can be found in [7]). And let have another state S_{new}, which is going to be loaded into the system.

Our proposed access rules are the following:

- **Complete reject.** This rule is based on the impossibility to create a collision with existing states ($S_{planned}$), either current or planned. Simply, if there is the collision, the transaction is aborted and the new state (S_{new}) cannot be inserted, at all – Fig. 1.

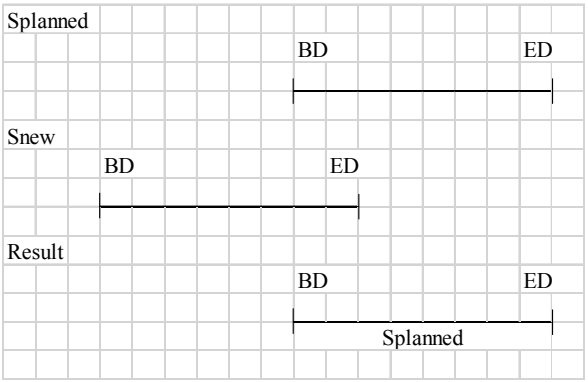

Fig. 1.Complete reject

- **Complete approve.** In that case, state $S_{planned}$ is canceled and replaced by S_{new}. This rule can be used, if there is no validity interval collision between states, as well. This option can be even divided into two categories regarding the collisions, which are delimited by the parameter *collision_only*, which can hold *true* or *false* value. If the *true* value is set, planned states are not influenced, if there is no validity collision. Vice versa, if the *false* option is set, all states, which begin point of the validity is greater than actually inserted, are removed from the current image, respectively are signed as invalid – Fig. 2.

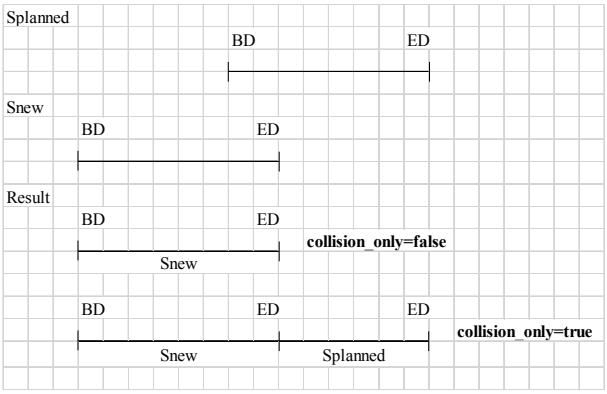

Fig. 2.Complete approve

- **Partial approve.** This approach principle is based on the shortening validity interval of the new state S_{new}. The validity end point of the state S_{new} is delimited by the planned state $S_{planned}$, which is signalized by the collision detector – Fig.3.

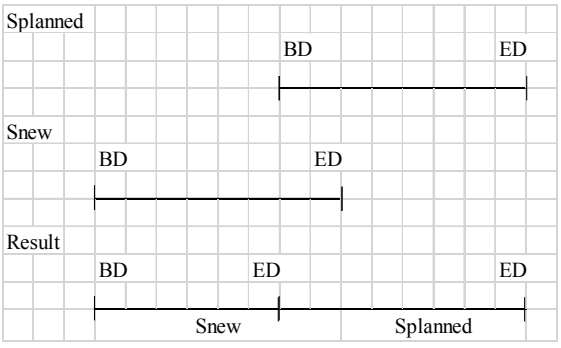

Fig. 3. Partial approve

- **Reposition.** It is based on the principle of remaining the new state original (as defined). Planned state $S_{planned}$ validity is either shortened, or the whole state is shifted starting at the *ED* of the inserted object state (S_{new}) – fig.4. The difference between *Complete approve rule* is based on the validity time frame. In case of using *Reposition*, original validity is retained, just the state is shifted in time, whereas *Complete approve* rule challenges by shorting the validity.

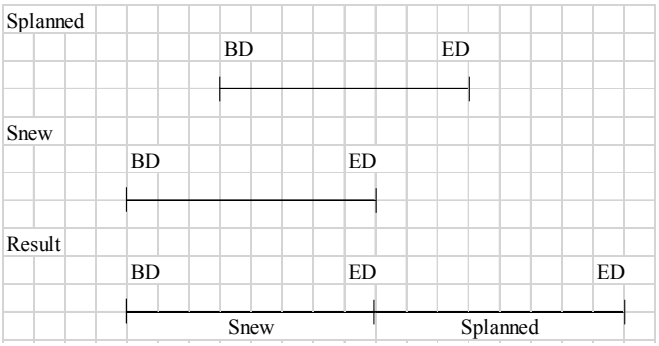

Fig. 4. Reposition

In the previous part of this section, we proposed solutions for dealing with future valid states, to solve the collisions with emphasis on the defined access rules. The management is based on the fact, that new inserted state influences only future plans, states, which validity will start later in the future. This sub-section proposes techniques for dealing with states, which has been valid either completely (historical state) or partially (current). In that case, solution management is a bit more complicated, due to the necessity to ensure extended integrity. Antidating problem definition is based on the fact of influencing the existing, non-planned state. Therefore, it is almost always necessary to solve not only states themselves but also configured and calculated outputs. If they are stored directly in the database, they can be distinguished and marked as non-actual. If the output result set or report is stored outside the database, it is not possible to message it. For such reasons, API functionality has been defined by us, which checks the relevance and appropriate values to be processed in the functions. Thanks to that, if the data portion or structure is changed, it is clear, that the calculated results are not based on current data image (delimited by the time interval or by the object region) and output does not need to produce correct data sets. However, the point is, how to detect it without the necessity or recalculation, without the necessity to execute such functionality once again and compare result-sets. Reflect the situation, that the processing can be really resource-demanding lasting too much time, thus it is not convenient, even possible in some conditions, to recalculate results just to detect, whether produced data from the previously executed function are the same as current or not. Therefore, in the past, antidating – changing data valid in the past – was prohibited. As a consequence, it was not possible to change any state inserted in the past, if the validity has already started. The reliability of the solution was lowered, whereas it was not possible to correct the state if the new conditions occurred. Some solutions used auxiliary data structures to store data about corrections, however, the main storage does not have information about corrections [4] [16]. As a consequence, it can be said, that the main system stores incorrect data. It is not reliable and secure, isn't it?

For the purposes of the change and correction detection, our proposed solution uses signature hashes, which are calculated for each object, each state, and the time spectrum, as well. Result set of the function stores a signature hash for the data set evaluated inside. Thanks to that, if the signature hash is changed later, it is just necessary to check and compare the database signature hash and a hash of the data provided as the input for the data object function. Principles and performance impacts are later described in chapter 3.

III. DATA HASH AND POINTER

Detection and correlation of the existing state changes executed later is a core part of the processing forming a reliable solution. The security aspect is delimited by the correct data image provided to the other systems as result sets. Evaluation of the particular states and identification of individual attribute changes can be a really complicated process. First of all, data can use various granularity types, from objects, up to attributes, respectively groups. If the object architecture is used, the original value can be copied multiple times to the new images, if no change of the particular attribute occurs. Another problem arises if the data from the sensorial network are collected periodically. In that case, the new value is produced, however, it does not need to express and reflect a real change. Therefore, it is necessary to evaluate and distinguish real and significant changes, even after using Epsilon techniques [8] [9]. Data attribute does not need to be size - small or atomic, it can be defined as a compound value with references or objects, which shift the problem deeper. A comparison of the attribute values themselves does not provide sufficient power, whereas it could be really extensive referencing other objects, large files, and structures. Therefore, in our proposed solution case study, a hash value for each relevant category is produced. The hash value can be calculated either for the whole state, for the attribute group, object or the whole image of the database, while being ensured, that any change causes generating the different value of the hash (security aspect of the solution). Thanks to that, change of the object, attribute or group can be detected easily. Moreover, if the object state is fluctuating and their values are the same with its historical image, the same hash values are produced, thus periodicity can be detected, as well. As a consequence, data can be compressed pointing to the cyclical sequence by highlighting only anomalies. However, as it may not seem at first glance, our proposed solution must encounter and solve many problems, which are described in the following sub-sections.

An Indexing and distribution sorting

Our produced hash values are obtained by the effective, but also mathematically sophisticated methods, to ensure unique value for any object, state or group. Thus, they can be used as row identifiers or locators. Moreover, whereas data type of the hash is *RAW*, data manipulation and comparison is really fast [15] [16]. On the other hand, the robustness of the hash

function causes approximately uniform values distribution over the whole set, whereas each hash function result set is delimited by the range bordered by the function *Mod* (the remainder after division). Any, even a small change of the value, should generate a completely different value. It means, that the states and corrections of the individual objects are distributed across the whole set. Therefore, the changes are not interconnected in any way. On the one hand, this is perfect from the point of the changes, on the other hand, obtaining individual images and producing object monitoring over time is difficult, whereas object images are difficult to be located in the database - they are widespread. To sharpen the problem, consider the index to locate data. If the hash is used for the individual states of the object, the data location is based on traversing the B+tree (as the default index structure for the database). Thus, if only one object state is used, the *Index Unique Scan* method is used, based on the precondition of storing unique hash values. If we want to get the states during the defined time interval, the optimizer must be based on the prediction, statistics, and pre-calculations, decide, how to access data. If only one state is produced, the solution is clear – defined index provides the perfect solution to locate data. However, if there are (or can be) several state modifications, the system must predict its number and decide, whether multiple time index access is more optimized in comparison with accessing the whole table consisting of all states of the object. Based on our previous computational studies [8] [9], we came to the conclusion, that such a solution is not fair. Although the index access and data location can be done in parallel, it does not provide sufficient power due to resource consumption and the necessity to synchronize data to the output packages. Therefore we create and compare several techniques to improve the state of the art. The current solution is based on multiple time scanning of the index. If three states are defined for the particular object, all of them are distributed with no pointer to the consecutive state. Thanks to that, the defined index is scanned as many as the number of states, we want to get.

B Pointer stitching

Index access using hash values is not optimal, whereas the index traversing must be done multiple times. It can be partially eliminated by the parallel data access, however, there is still point of the synchronization of the result set and index locking. Therefore, we extend the B+ tree definition by using stitchers across the whole structure, thus, individual hashes are not linked together in the leaf layer, whereas its sorting does not provide any benefit/hash values do not provide direct data information. Instead, pointers are directed to the consecutive state of the object. Thus, data are not interconnected using hash values are an identifier, but another layer consisting of pointers to individual states of the object is used. These states are sorted in the time interval position manner. If the state is corrected, a particular object is locked inside the index and arrows are recalculated. Thus, historical (original) states, which were later changed (corrected) are not part of the index.

If the object composition is present, additional pointers to the other group set members are present.

Bi-temporality can be modeled using the already described solution, as well. Timestamp expressing insert date is added to the system expressing the transaction layer (the first layer is validity, the second temporal layer is a transaction – bi-temporal approach). Individual corrections are stored separately sorted by the transaction frame, but connected to the main index structure using the pointers. In general, pointer correction originate is the object itself.

C Object grouping

The limitation of the last described solution is just the heterogeneity and distribution of the data across the index. One object is split into individual states and managed separately. On the one hand, the structure is relatively easy to be managed, on the other hand, states for all objects are present in one index structure. Although the B+tree index does not degrade with the number of blocks and levels, the important side is just the size and severity of the stitching. Therefore, inspired by the *Index Skip Scan* access method, we introduce index in index access pointers. The architectural solution is based on the master index managing only object headers, not the specific data states for them. Each object is identified by its own unique signature hash, which is part of the index. Thus, on the leaf layer of the index, there are not only data locators inside the database, but we also put their signature hash. Individual states themselves are then connected via a secondary index, similar as described in section 3.2. The interconnection between these two layers is done using a nested table structure consisting of individual object state hashes. Thanks to that, identification of the state changes across the defined time interval can be done directly at the master level (if some change even occurred). The details of the change itself, respectively attribute images of the object are then in the second layer. Fig. 5 shows an overview of the architecture.

D Locating change inside the hash value

The hash value itself does not have any specific structure, by which individual attributes with changed values can be identified and consecutively located. From the object outside point of view, it is evident, that particular change occurred, however, it is not possible to determine, which attributes have different values in comparison with previous states. In this evolution step, it is necessary to split the state into individual attributes and compare individual attributes to each other. Moreover, if one attribute changed its value also others could change them, thus the whole attribute set must be scanned, either sequentially, or by using parallel techniques. Moreover, conventional and static attributes must be taken care of.

For the optimization of the locating change process, signature hash values are extended. We propose a multi-module solution. Signature identification consists of three parts:

- the calculated hash value of the state, the calculated hash value of the object (for the reference of the master index),

- nested table structure holding identifiers of the attributes, which values were changed reflecting the direct previous change.

Thanks to the defined structure, consecutive changes can be identified and located relatively easily. If two independent states are to be compared, sequential attribute scanning is necessary. However, how does the nested table structure look like?

We have developed three solutions, which are compared in the performance section, as well:

- Nested table stores list of identifiers of the attributes, which were changed *(model A*). It means, that the size of the nested table depends on the number of changed attributes. As a consequence, the size of the structure is variable, thus each database block can consist of a different number of nested structures, which must be scanned sequentially, whereas there is no order. The size of the structure is not constant resulting in free space in the block.
- The second proposed solution (*model B*) is characterized only by the number of changed attributes for the state. It stores only one integer number. Internally, individual attribute scanning is ordered based on the statistics and prediction of the attribute to be changed. These extensional statistics are obtained in the maintenance windows periodically. The relevance and effectivity are the same as ordinary statistics – if huge data are changed or loaded, statistics are inevitable to be recalculated to represent the current situation. Our own extensional statistics management is connected to the standard types using internal database jobs.
- The last proposed solution (*model C*) uses the attribute change identifier (*chID*). The first part of the *chID* is calculated from the attribute values, which are unique and deterministic from the definition. The second part is the link to the attribute identifiers, which hold changes. It is represented by the integer value, the transformation and denotation itself are stored in the code list database table. Thus, the size of the structure is always the same, regardless of the number of changed attributes. On the other hand, originally, there was a significant disadvantage, if the new attribute of the object is added to the temporal system, code list values must be recalculated. In our solution, however, it is done online by using temporal evidence and managing transformation automatically without user intervention necessity. Therefore, any user mistakes resulting in incorrect transformation and expression are removed.

Own proposed solutions in this paper are based on the validity management and monitoring. In chapter 2, reliability and collision management rules were proposed forming another layer controlling data to be inserted and indeed accepted states. In the future, our research in this area will focus bi-temporal architecture evaluating data corrections. Change identifier is significant for the consecutive processing in calculations, function results, reports, etc. We will extend these techniques with the aspect of data correction. By using it, we suppose more robust solution identificating specific change and reflection to pre-calculated and pre-processed outputs.

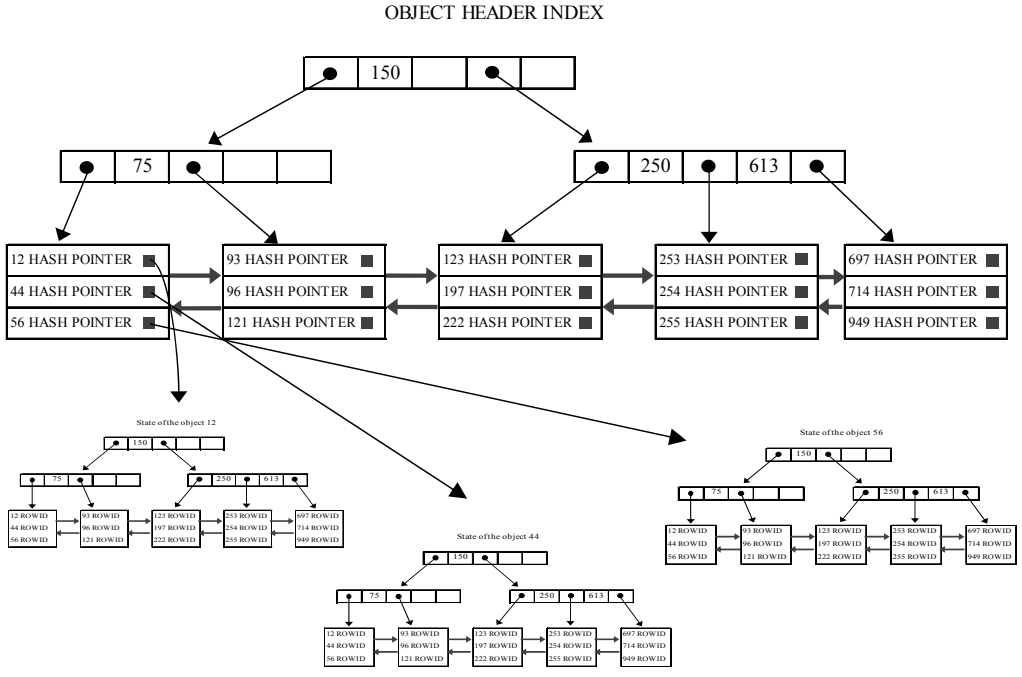

Fig. 5.Approach illustration

IV. PERFORMANCE

Experiment results were provided using Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 64bit Production; PL/SQL Release 11.2.0.1.0 – Production. Parameters of used computer are processor: Intel Xeon E5620; 2,4GHz (8 cores), operation memory: 16GB, HDD: 500GB.

Environment characteristics are based on a real environment consisting of *1000* sensors producing data *ten times for one minute*. 10 percent of the provided data is consecutively replaced (corrected) by newer ones using versions. Three models described in chapter 4 are used for the evaluation. *Model A* stores pointer to the attribute changes in the linked array. *Model B* stores the number of changed attributes, which are then accessed. Thanks to the number of changed attributes, searching can be stopped, if the requested amount is reached. Improved model (*model BB*) stores the attributes in the sorted list based on the frequency of the changes covered by the prediction techniques and statistics). *Model C* uses *chID* and references code list. Fig. 6 shows the performance represented by the processing time, resource consumption and size demands. Individual solutions are compared with the pure existing solution *Model X* with no validation characteristics of the change using hash (reference model – 100%). For the evaluation, results are expressed in percentage.

Performance evaluation can be done in three layers. When dealing with processing time, the best solution reached *model C* and *model A* with the processing time improvements using 20% (reference *model X*). *Model B* reached improvement using 5%, *model BB* using 8%. In models B and BB, it is necessary to evaluate the whole nested table, regardless of the defined attribute group to be evaluated, whereas it has no specific order, nor the access index can be defined. When dealing with resource consumption, the best solution provided *model C* (improvement 36%) and *model A* (35%). The principle of the requirement is ensured by the direct access to the attributes, which values were changed. In *model A* – it is done using nested table array, *model C* uses a pointer to the code list managing the attribute change itself. The last evaluated criterion is size demands. Whereas for each proposed solution, additional structure is added, size requirements must be higher. For *model B* and *model BB*, it reflects the change using 2% - only one integer value is added. For *model C*, also code list is stored in the file system providing all data attribute combinations. It requires an additional 5%. Model A stores the whole list for each object state, therefore the requirements are increased using 70%.

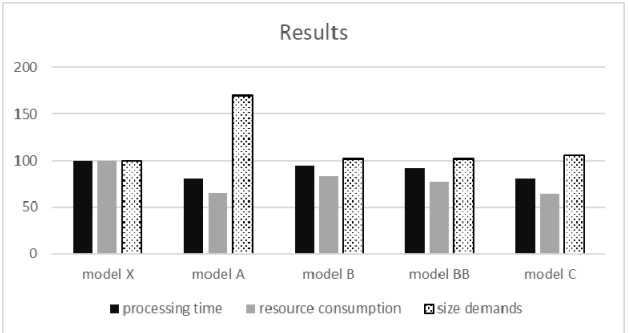

Fig. 6.Performance characteristics - results

V. CONCLUSIONS

The conventional paradigm is based on storing only current valid data. Although historical images can be partially obtained by using log files, it is a really complicated process requiring huge resources and processing time. The temporal extension can manage data during the whole life cycle of the object storing all states. In the past, several approaches have been proposed to deal with the temporal paradigm using the validity aspect modeled by the time frame. The object, attribute and synchronization group granularity can be used. The limitation of such a solution is just the state changes identification and location. If the attribute value is obtained periodically, or object granularity is used, the same values characterizing an object or its part can be reached. In that case, particular data are not stored (if possible) – existing value validity is shifted or duplicate value is stored. Anyway, if the object state changes and evolution must be obtained, such states must be excluded by highlighting only significant changes. For such purposes, we propose a hash definition, which can be placed for object, attribute, group, defined database image during the defined time frame. Thanks to that, change identification can be done easily, it is possible to ensure reliability and security aspect, as well. Using extensions proposed and described in this paper, the solution provides an efficient approach locating changes up to attribute granularity. Thus, transaction validity can be reached by using a master and secondary index based on the pointer layer and object grouping, as the best solution defined by the performance relevance. At the same time, we have solved the issue of hash value distribution inside the index, so that the individual object states are linked using two directional linked-lists.

Many times, database management requires a distributed environment and balancing of the individual nodes. In the recent future, we would like to extend our solution in a distributed environment manner to cover the complexity of data distribution. Each data tuple is then located on at least two data nodes to ensure data access in case of node or sub-net failure. Surviving data nodes take care of the management and redistribute data, so the data are available on at least two nodes then, again. A similar principle will be used for adding new nodes to the system, as well. Dealing with the transaction management inside the temporality, it requires to locate data on the particular nodes and ensure data corrections, if the existing state is to be modified. For such purposes, signature hash values with pointers to individual nodes would be used.

ACKNOWLEDGMENT

This article was created in the framework of the National project IT Academy – Education for the 21st Century, which is supported by the European Social Fund and the European Regional Development Fund in the framework of the Operational Programme Human Resources.

The work is also supported by the project *VEGA 1/0089/19 Data analysis methods and decisions support tools*

for service systems supporting electric vehicles and *Grant system UNIZA*.

"PODPORUJEME VÝSKUMNÉ AKTIVITY NA SLOVENSKU PROJEKT JE SPOLUFINANCOVANÝ ZO ZDROJOV EÚ"

REFERENCES

[1] Ahsan, K., Vijay, P., 2014. Temporal Databases: Information Systems, Booktango.

[2] Ashdown, L., Kyte T., 2015. Oracle database concepts, Oracle Press.

[3] Avilés, G., et al., 2016. Spatio-temporal modeling of financial maps from a joint multidimensional scaling-geostatistical perspective. In Expert Systems with Applications. 60, 280-293.

[4] Doroudian, M., et al., 2016. Multilayered database intrusion detection system for detecting malicious behaviours in big data transactions, IEEE International Conference on Industrial Engineering and Engineering Management (IEEM).

[5] Erlandsson, M., et al., 2016. Spatial and temporal variations of base cation release from chemical weathering a hisscope scale. In Chemical Geology. 441, 1-13.

[6] Johnston, T., 2014. Bi-temporal data – Theory and Practice, Morgan Kaufmann.

[7] Johnston, T., Weis, R., 2010. Managing Time in Relational Databases, Morgan Kaufmann.

[8] Kvet, M., Matiaško, K., 2017, 5.7. – 7.7.2017. Temporal Data Group Management, IEEE conference IDT, 218-226.

[9] Kvet, M., Matiaško, K., 2014, 18.6. – 21.6.2014. Transaction Management in Temporal System, IEEE conference CISTI, 868-873.

[10] Kvet, M., Matiaško, K., 2014, 20.11 – 22. 11.2014. Uni-temporal modelling extension at the object vs. attribute level, IEEE conference UKSim, 6-11.

[11] Kuhn, D., Alapati, S., Padfield, B., 2016. Expert Oracle Indexing Access Paths, Apress.

[12] Kumar, N., 2019. Efficient data deduplication for big data storage systems, In Advances in Intelligent Systems and Computing, 714, 351-371.

[13] Li, S., Qin, Z., Song, H., 2016, A Temporal-Spatial Method for Group Detection, Locating and Tracking, In IEEE Access, 4.

[14] Li, Y., et al., 2016, Spatial and temporal distribution of novel species in China, In Chinese Journal of Ecology, 35, 7, 1684-1690.

[15] Yu, C., et al., 2019. A fast LSH-based similarity search method for multivariate time series, In Information Sciences, 476, 337-356.

[16] Zhang, K., et al., 2019, Efficient public-key encryption with equality test in the standard model, In Theoretical Computer Science, 755, 65-80.