

Temporal Data Version Identification using Signature Hash

Michal Kvet, Karol Matiaško

Department of Informatics, Faculty of Management Science and Informatics

University of Žilina

Žilina, Slovakia

Michal.Kvet@fri.uniza.sk, Karol.Matiasko@fri.uniza.sk

Abstract—Current applications and information systems are data oriented. Temporal characteristics covers the core principle of information system intelligence. Validity frames each data state and the whole evolution is stored allowing complex analytics, decision making and processing. Management and evaluation can use various granularity models. This paper deals with uni-temporal architecture using object, column and hybrid precision. It proposes new technique for dealing with existing data tuple updates forming new versions. Whereas historical state versions could be used, reliability of the whole system can be compromised. To deal with individual versions, we propose signature data hashing to identify evaluated data image.

Keywords—temporal signature hash; data version; temporal granularity model; hybrid synchronization; reliability

I. INTRODUCTION

Intelligence of the current information systems is based on the data, which are evaluated, managed, stored and analyzed. It is inevitable, that data are stored during the whole life cycle of the object, handling data changes in the past, as well as current and future valid tuples. To get the performance benefit, it is necessary to optimize database structure, choose relevant approaches and techniques for data management in the structure and propose robust access methods for queries.

Data entering the system are heterogeneous with diverse structure, characteristics and properties. Therefore, database architecture must be able to handle them in effective manner with emphasis on the temporal aspect. The first part of the paper summarizes the temporal evolution and describes models for dealing with temporal data with emphasis on granularity. These data are made of objects, which are composed by individual data tuples bordered by the validity time frame. The term validity is significant, whereas it is an element for ordering states and proposes the ability to track evolution. It is usually not possible to change existing state, to add new object state replacing already stored in the database or to limit the validity of existing tuple. Second part of the paper deals with the techniques of managing data corrections. The aim of that part is to propose own solution module for storing, evaluating and managing data versions without necessity to create bi-temporal solution. Thanks to that, any data tuple can be later replaced by the newer state based on changed conditions. We keep not only validity images in the time, but also data changes, corrections and modifications. Our proposed solution

is based on the data signature hash, which makes it easy to identify the version of the database used for a particular analytical operation or aggregation.

II. RELATED WORKS AND TEMPORAL EVOLUTION

Temporality as a core building element has been considered to be dominant since the first database system attempts. Data were bulked into the transactions to ensure the consistency of the data and reliability of the whole system [11]. The basis of the transaction is the log management secured by the *Log Writer (LGWR)*. Each data change is registered in the physical storage log file consisting of date point of the execution, before and after data image [1] [2]. Whereas transaction ensures no data can be lost, by using log files, historical data can be obtained. The necessary condition is to keep all the log files accessible in the file system. Nowadays, it is not a significant problem, because disk space prices are low, regardless the infrastructure – physical hardware or cloud storage. However, in the past, there were problems with historical data log file detection. To provide historical data image, all log files had to be scanned sequentially followed by the identification of the change of the particular object. If some log file was lost, there was no evidence in the system, which was later partially solved by the implementation of the *System Change Number (SCN)* in the header of the log file. Each database change incremented the value of *SCN*, thus, undefined or missed states could be detected. Main disadvantage of such approach was the efficiency of the whole system. Sequential scanning is really time and resource demanding without any guarantee that any change was made [10] [18].

More advanced techniques of log file management are based on the flashback technologies [9] [10] [16], parameters of which determine the object evolution during the defined time point or interval. Another aspect is based on obtaining consistent image in the defined time in the past. On the one hand, this system is automated, however, still too resource demanding for the results to be used in real world, to be the basis of the complex decision making [8]. And the limiting factor is just the data background – it can deal only with current and historical images. There is no space to cover also future valid data, which were processed externally in the first phase. Thus, to cover the whole time spectrum evolution, interval log files had to be scanned, followed by current image and module for dealing with future states, which were stored

externally. It means, that the main structure did not have a complex information about changes in the future. Also, external module had to ensure new valid state reflection in time, which often caused inaccuracies, delays and last but not least, the inconsistency [17]. Previously defined solutions can be named as logical, whereas there is no change in the data structure to provide historical images in comparison with standard conventional practices.

These limitations led the developers and researchers to create new temporal paradigm by extension of the database structure. In 90ties of 20th century, new temporal paradigm was created. It was mostly based on the object primary key definition extension by the validity frame. Thus, primary key of the object was composed from the object identification itself and validity time frame modelled by the time interval (BD , ED) or just one attribute characterizing start point of the validity (BD). In that case, each newer object value (state) automatically ends validity of the previous state. Interval and time point transformations can be found in [9] [10]. Fig. 1 shows the architectural models – conventional model without time definition, uni-temporal solution characterized by the time interval (BD , ED) or compressed uni-temporal solution with only BD time value. Such defined models are object oriented, thus any data change automatically creates new complex object image tuple. As a consequence, it can generate huge amount of the same values (duplicates), if the attribute value is not stored at each change (update). Although it is partially possible to remove such impact, it is not markedly optimal.

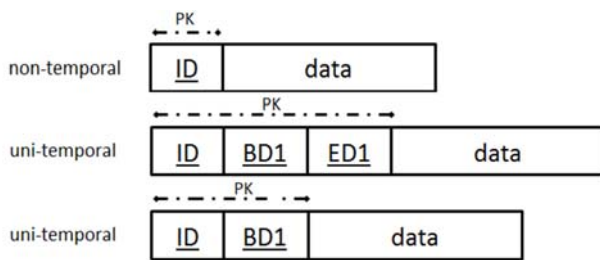


Figure 1. Uni-temporal validity model using object granularity

Attribute (column) oriented approach has been proposed in 2015 and is based on column granularity. Each data change is divided to the individual attributes, which are managed separately. Thanks to that, there are no duplicate values. If the data value is the same as direct predecessor, particular value of the validity is not changed and original value is kept [15].

Architecture of attribute oriented approach is shown in the fig. 2. It consists of three layers, one of them, managing historical and future valid images is internal, thus, it cannot be queried and processed directly. Current valid data are located in the first layer. Core part of the system architecture is in the second layer forming temporality. Each change is registered in the temporal module. It ensures the automatization of data changes – if the future valid state becomes current, reloading is done immediately. In comparison with other systems, this solution is based on database layers, thus there are no delays between planned begin time point of the validity and real change of the execution. Therefore, there is no reliability risk based on storing and evaluating non-actual data [11].

Moreover, whereas current valid data are stored separately, if the existing application deals with conventional approach by managing only current valid data, they can continue to work without changes, without necessity to rewrite code, to recompile solution. Just the database connection string is changed, if ever.

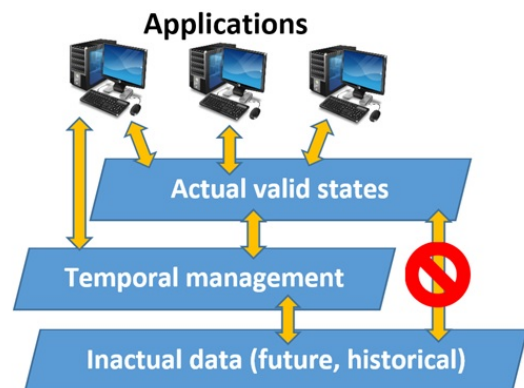


Figure 2. Attribute oriented approach

Attribute oriented granularity is effective regarding duplicate values. It forms the opposite of the object granularity, where each data change creates complete new data tuple. On the other hand, if some data changes are synchronized, it requires new Insert statement to the temporal module for each attribute, which is performance limitation, as well. Therefore, in 2017, hybrid solution has been proposed, which extends attribute oriented granularity approach by the definition of synchronized attribute groups, which can be created and managed manually or automatic identification can be used. Such solution consists of six layers, added layers are monitoring changes, register new synchronized groups, alter them, if necessary or remove them, if the balance is not suitable. The aim of such module is to minimize disk storage space, to optimize performance and reduce system resources of the processing. Fig. 3 shows the architecture of hybrid temporal solution [13]. If the group is detected, first layer (*detector*) sends the message to the second layer (*manager*) and asks for creating the group. Manager creates the group and notifies the *detector*. Afterwards, individual attributes are added to the group forming extension. Consequently, particular data are formed and rerouted from the database layer itself to the synchronization layer, which has these tasks:

- Synchronize input data values and form the batch.
- Load the batch into the database.
- If data are coming asynchronously, it must notify group *manager* to evaluate the situation.

Some attributes cannot be grouped at all due to the data value structure or due to data character. Such information can be explicitly defined in *group manager layer*, which ensures such functionality by notifying *detectors*.

Data group synchronization module can be located either on the server directly connected to the database or externally on the client of intermediate site. The main advantage of server location is direct database access supported by new background processes (*Synchronization Manager*). Remote synchronization

module queries the database and evaluates the impact of the new or existing group. Connection is ensured by new *Remote Synchronization Manager* protecting server process and session itself. Fig. 3 shows the architecture of server site solution, fig. 4 shows external group detection module.

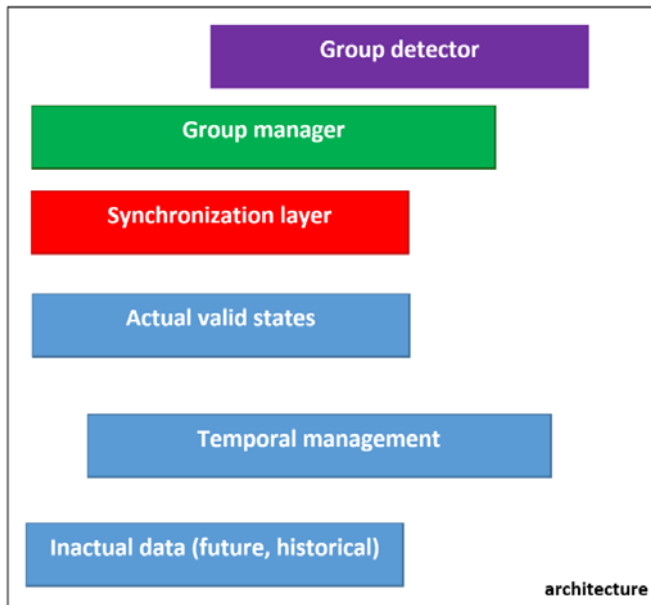


Figure 3. Hybrid temporal architecture [13]

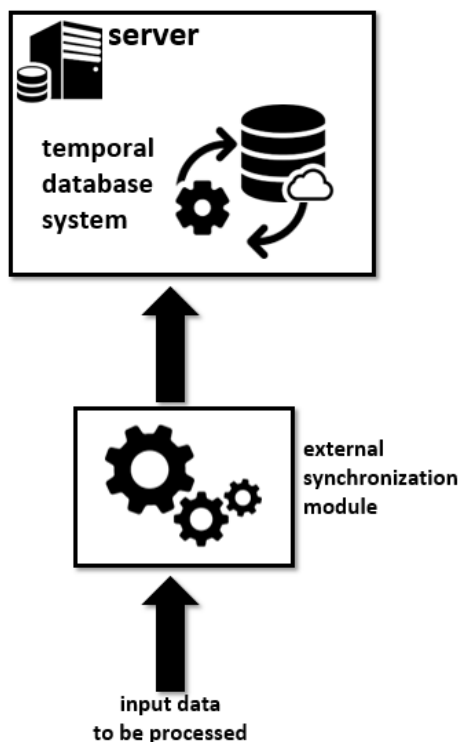


Figure 4. External synchronization

Other approaches dealing with time definition are based on locality extension of the temporality forming spatio-temporal

databases [3] [4]. Significant research stream is formed by the non-relational paradigm [6] [7]. In that case, however, data are in raw form with almost none transaction control.

III. DATA CORRECTIONS AND ANTIDATING

The main limitation of the previously mentioned approaches is just the modelled time spectrum. It deals only with the validity temporal aspect. Thus, each data state (tuple) is bordered by the validity time frame, which can be modelled using various models. It can be defined as opened or closed for the left and right site of the interval, as well. Thus, in principle, validity interval can be modelled using the following types:

- Closed – closed representation
- Closed – opened representation
- Opened – opened representation
- Opened – closed representation

Based on complex modelling and representation, begin validity point is almost always represented by the closed characteristics to simplify change identification. The main reason arises from the future valid data management, where timepoint of the transformation must be precisely specified. If not, delays or inaccuracy can be identified. Individual transformation of closed and open characteristics can be found in [10].

Individual states can be sorted or treated positionally in the time spectrum. For the evaluation, Allen relationships can be used [10]. Fig. 5 shows the potential of the relationship description.

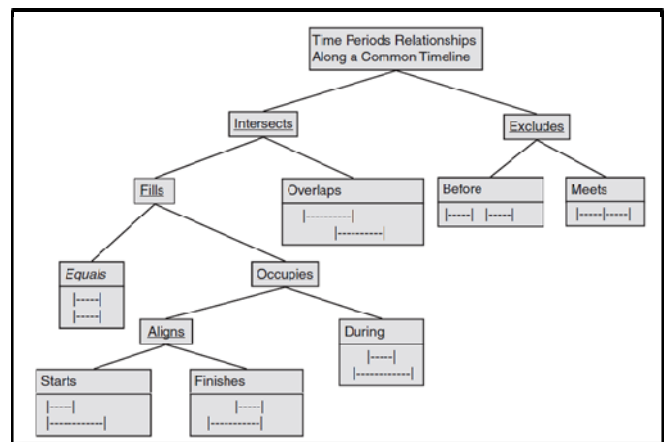


Figure 5. Allen relationships [10]

In temporal databases, three situations are mostly highlighted:

- *Intersects* ensuring any data image is not modelled by more than one valid state. Simply, one object can be modelled by no more than one valid state anytime. It covers the alternatives of the interval overlapping or filling (intervals are the same, have

the same begin or end validity point or one of them completely covers the second one)

- *During* relationship is used for the referential integrity.
- *Excludes* relationship (covering *Before* and *Meets*), which is used for the evolution management and sorting. *Meets* relationship express, no other state can be located between such states.

As already mentioned, such models deal only with the validity. If the already existing state is to be updated, original state is removed, respectively updated. As a consequence, database stores only data currently considered to be true – as they were, are or will be, without any evidence of corrections and changes. Objects are characterized by the evolving states, however, individual data state versions are not managed at all. Fig. 6 shows the problem of data update. Let have the object *O1* identified by two states *S1* (valid from *T1* up to timepoint *T2*) and *S2* (valid from *T2* up to timepoint *T3*). If the state already stored in the database is to be updated, several approaches and situations can be used. The simplest and, unfortunately, most often used principle is based on prohibiting data modification. This principle is called antdating prohibition. As a consequence, whereas data cannot be versioned and corrected, database stores non-reliable data and analytics can be corrupted, with no evidence to the client site. In contrast, another principle is based on allowing any change. If it reflects future valid state, there is no problem, whereas it has not been valid, yet. Current and historical data are important, because they could have been used and processed in analytics, decision making support or simply copied to archive destinations [12].

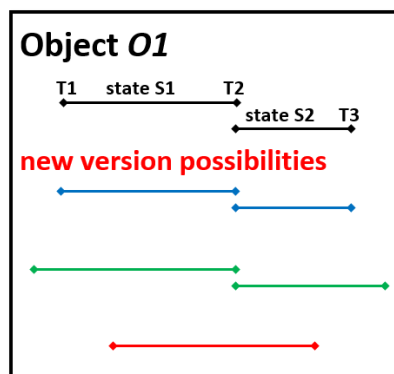


Figure 6. Possible data changes

Fig. 6 shows two existing states (*S1*, *S2*) of the object *O1*. If new version is loaded into the system, three possibilities can occur:

- validity is not changed (blue part),
- validity is extended (green part),
- validity influences more than one existing state (red part).

Transaction management of state modification can be covered by transaction rules described in [14]. In general, existing object state cannot be directly modified.

IV. VERSIONING

Solutions based on uni-temporal systems cannot manage data versions and do not store all data flow with emphasis on data corrections. One of the possible solution is based on extending uni-temporal by storing also transaction time, which characterize period, during which object was considered to be true. Such solution is called bi-temporal (third model, fig. 7). As a consequence, referential integrity is far more complicated to ensure complete covering of reliable data [5].

Let point out also to the one important aspect forming reliability of the whole system. How would you recognize, which data version of the particular state has been used for the processing and evaluation. In principle, it should be the latest one based on date of the loading execution and date of the analytical process start. In the distributed environment, it is far more complicated to determine it. It is therefore necessary to store not only transaction date period for the main transaction, but also processing time for the particular database node. Thus, three-temporal architecture is used – validity (*BD1*, *ED1*), transaction validity of the global transaction (*BD2*, *ED2*) and transaction validity for the particular web node (*BD3*, *ED3*). In general, multi-temporal solution can be used (the fourth model, fig. 7).

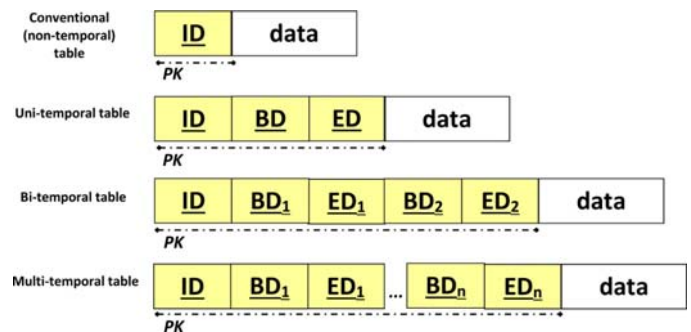


Figure 7. Bi-temporal and multi-temporal model

V. OWN VERSIONING SOLUTIONS

Data versioning management is inevitable part forming reliable solution. The ability to cover all the state versions shifts the solution to be the robust, however, it is only one part of the problem complexity. Generally, data stored in the database are monitored, evaluated and analyzed based on current data in the reliability manner. Thus, if some result is provided, it must be clear, which data image was used, if multiple versions are available. Our environment was created using Oracle database system, thus no data phantoms can occur. On the other hand, it is useful to keep in mind processed data versions, whether the query deals with the data accessible at the begin time point of the transaction or only query itself. To have the relevant answer pointing to the data image evaluated, we propose solution based on the signature hash described in the section 6. However, before the evaluation itself, our proposed architectures and research streams must be described.

The first proposed solution of this paper is based on the versioning extension of the uni-temporal solution. Index pointers are always routed to the newest version, thanks to that, each new analysis uses current version image at particular execution moment. Historical versions are stored in the nested tables of the same temporal structure. Fig. 8 shows the architecture of the solution using object granularity. In principle, any granularity can be used. Relevant data state is encapsulated by the validity time frame (*BD*, *ED*) and insertion date (*IND*). These data reflect the newest version of the state. Insertion date is recorded automatically using *sysdate* (date and time in the second granularity) or *sysimestamp* (date, time, up to 1ns precision) function, or any analogous function for the used database system. If there is no previous version for the state, particular nested table is empty. If the new version for the particular state is inserted, original is transferred to such nested table and main structure continuously stores only top version. Therefore, it is necessary to distinguish between empty nested table (in that case, before the transfer, constructor function must be called) and existing nested table with previous state versions.

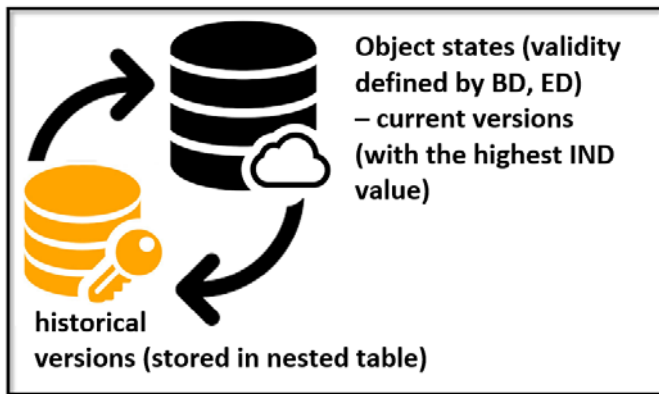


Figure 8. Nested table storing versions

Versioning storage in this defined solution can, however, form several state collisions, which is demonstrated in the fig. 9. Let have one existing state (*S1*) consisting of only one version. After the processing, state *S1* is modified. In that case, new version for the state *S1* is created. In principle, two situations can occur. In the positive change scenario, new version does not change the validity time frame (fig. 9, part A). More complicated situations occur, if the validity frame is changed. In that case, it is not sufficient to create only new state version, original state must be divided into two or three ones. Let assume, that the state *S1* is validity bordered by the attributes *BD1* and *ED1*. Afterwards, new version is created with the validity time frame delimited by the *BD2* and *ED2*. Three situations can be identified:

- $BD2 < BD1$ and $BD1 < ED2$ and $ED2 < ED1$ (fig. 9, part B). In this case, original state must be splitted into two parts and one new version is added:
 - State framed by the *BD2* and *BD1* is delimited by new version.

- State framed by the *BD1* and *ED2* is formed by the original version, which is also replaced by the new version.
- State framed by the *ED2* and *ED1* remains original (version is not changed, only validity interval is shortened from the left site)
- $BD1 < BD2$ and $ED2 < ED1$ (fig. 9, part C). In this case, original state must be splitted into three parts:
 - State framed by the *BD1* and *BD2* remains original (version is not changed, only validity is shortened from the righth site).
 - State framed by the *BD2* and *ED2* is formed by the original version, which is also replaced by the new version.
 - State framed by the *ED2* and *ED1* remains original (version is not changed, only validity interval is shortened from the left site)
- $BD1 < BD2$ and $BD2 < ED1$ and $ED2 > ED1$ (fig. 9, part D). In this case, original state must be splitted into two parts and one new version is added:
 - State framed by the *BD1* and *BD2* remains original (version is not changed, only validity interval is shortened from the right site)
 - State framed by the *BD2* and *ED1* is formed by the original version, which is also replaced by the new version.
 - State framed by the *ED1* and *ED2* remains original (version is not changed, only validity interval is shortened from the left site)

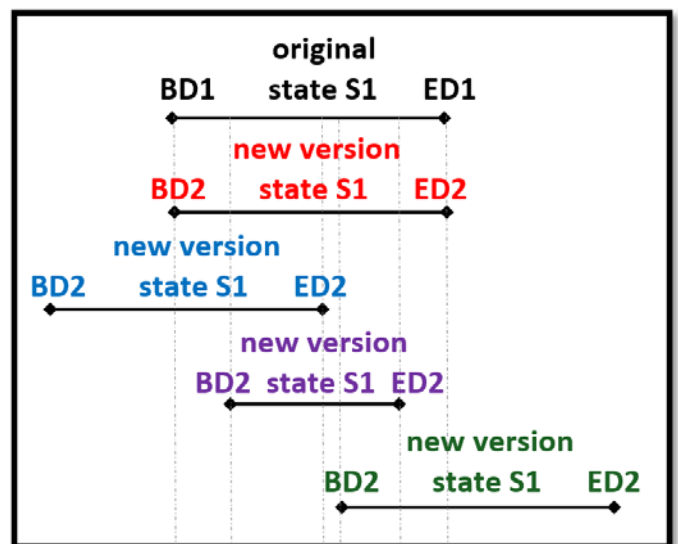


Figure 9. New version

Generally, new version can influence many states, however, only interval marginal parts (the top left and right) are splitted into two parts.

Second our proposed architecture divides current versioning from the historical ones by forming separate database storage. In that case, main structure is uni-temporal, the rest versions are stored in the historical database, which consists of the validity time frame (*BD*, *ED*) and insertion date (*IND*), as well. Versioning itself is covered by the triggers, which shift original version to the historical database repository. Consecutively, it is replaced in the main structure by the new version. Previously mentioned situations changing validity of the original state can occur in this architecture, as well. On the other hand, it is not necessary to deal with constructors and specific storage for nested tables. Triggers are associated with the destructive operations – *Update* (modification of the object by adding new state or version) and *Delete* (removing object from the system, either by the direct delete operation or by moving historical data to another repository, like data warehouse). *Insert* operation expressing adding new object in the system is not necessary to be triggered, whereas it always loads the first and only one version for each statement.

The third architectural solution is similar to the second one, but the trigger management module is replaced by the new background processes (*DBVern*, *n* expresses the number of such processes in the system) of the instance. They have direct database access, thus it removes the slight impact of the trigger firing. Moreover, performance benefits, because these processes are always accessible in the memory (information about their existence is written in the *spfile* and are created during the *mounting* process of the instance). In distributed environment, version processes are present on each node. Several *Database version processes* can be present in the instance, they can be either general (in that case, if new version is to be loaded, random free *Database version process* (*DBVern*) is selected to process the request. In practice, we select the process to ensure performance balance. Number (*n*) of version processes is dynamic, if there is no enough processes, versions are too much queued, system (using *SMON* background process) automatically creates new processes based on predicted future workload (evaluated based on the version statistics collected periodically). Another principle is based on the process association to the precisely defined object group.

The last fourth model proposed in this paper is the generalization and simplification of the bi-temporal architecture and deals only with validity and database insertion date. It does not cover the whole interval. In principle, each new version delimits the validity of the previous one. In comparison with previous solution described in this chapter, there is no necessity to split existing states into parts, only during the image reflection to the user, individual positional time intervals must be evaluated. As you can see in the performance evaluation, it can be the bottleneck of the system. It is based on the assumption, that the state version correlates the insertion date (*IND*). For the evaluation and sorting, analytical function *RANK* is used:

RANK() over (PARTITION BY state_id ORDER BY insert_date DESC).

Function *RANK* is analytical and for these purposes, it gets the serial number for each version. Each current version of the state gets the value *I*, historical versions are then covered chronologically using insertion date to the database. There should be no gaps of the obtained values from the *RANK* function. If there is some gap present, it means, that data are not consistent – object state is covered during some defined interval by more than one valid version, which is not allowed.

VI. USED DATA VERSION IDENTIFICATION

Functions, analytics and aggregations are based on current image of the database, which is evaluated. When result set is stored or provided to the user, it is not clear, which specific versions were available at the time, and which are were not present. Thus, although the data result set is provided, it is not transparent, whether the results are still usable, since it was possible, that some other versions were loaded later giving previously images non-reliable. Typical example can be prediction on the one site and real data processing on the second site replacing calculated (predicted) values by effective ones. Therefore, to determine input image of the processing, data signature hash is stored with each processed data result. From such value, it is easy to determine validity and usability of the results. Moreover, there can be automatized functionality executed either automatically or based on specific conditions to remove old function results. In our approach, each new data shifts the signature hash to another value.

Each change, regardless of whether it changes the state or affects the version of the existing state, is stored in the database. Evidence of a given event is in the temporal layer of a particular model with respect to granularity (object, column, or hybrid design). The change itself can, but does not need to change already processed data in analytical tools. It depends on the images entering the analytical module, whether they are also modified. Therefore, each object changes the signature of the entire system, it creates own and unique fingerprint of the change. It works like following. Let have the complete image of the database covered by the actual signature *SIGNact*. New state is added (*Snew*). Fingerprint hash is added to the actual signature *SIGNact* and the whole unit is signature hashed. Such value is then accessible in the system, therefore each function dealing with data gets the actual hash as the parameter, which is directly copied to the result set.

Whereas signature hashes are based on *SIGNact* value, used image is trivial to determine – individual executed operations are consecutively subtracted from the hash. Principles are shown in the following figure. Current data are covered by the *SIGNact* value, each new states replaces it. It is sufficient to store only actual value, the rest ones can be dynamically calculated based on provided data (which are stored in the temporal architecture, so there is always possibility to get historical data image).

The data management is sophisticated, on the other hand, if new data are loaded, but they do not influence already processed results (as the output of analytics, functions, etc.), new function result set are not created, whereas it would provide the same values.

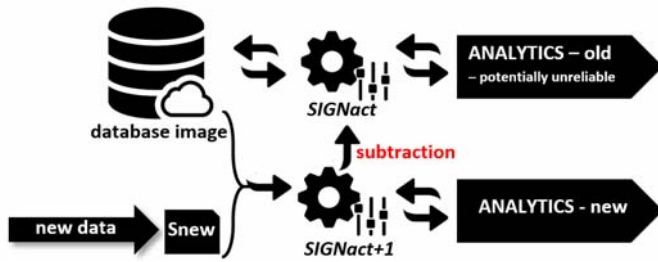


Figure 10. Signature hash calculation

VII. 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 real environment consisting of 1000 sensors producing data *ten times for one minute*. 10 percent of the provided data are consecutively replaced by newer ones using versions.

Five models have been used for the evaluation. The first one (*M1*) is based on original uni-temporal solution, individual versions are not processed, at all. Thus, approximately 10 percent of the processed data are unreliable. The second model (*M2*) deals with versions stored in the nested table for the particular state. Model 3 (*M3*) uses separate data structure (physical table) for dealing with historical versions. Model 4 (*M4*) is characterized by the background processes managing and accessing data versions. The last, fifth model (*M5*) uses bi-temporal architecture. Each state is delimited by the validity and reliability expressed by the time frame.

Tab. 1 shows the performance results with emphasis on the size for the whole structure and processing time - getting current image of the database with the latest versions of the individual states. As you can see, bi-temporal architecture reaches the worst results, for the size, as well as processing time. The reason is based on storing all data versions in the same table, thus the data amount is significantly rising. Although there are indexes to optimize data access, individual versions, as well as number of processed data, complicates the situation and huge data amount causing it widespread. Looking to the results, it can be concluded, that although size is increased using 9 percent, costs, CPU and processing time are increased by approximately 30 percent.

On the first sight, the best solution provides original uni-temporal solution (*M1*), however, it does not manage versions at all. As we can see from the experiment results, module for dealing with versions (*M4*) requires less than 7% increase of

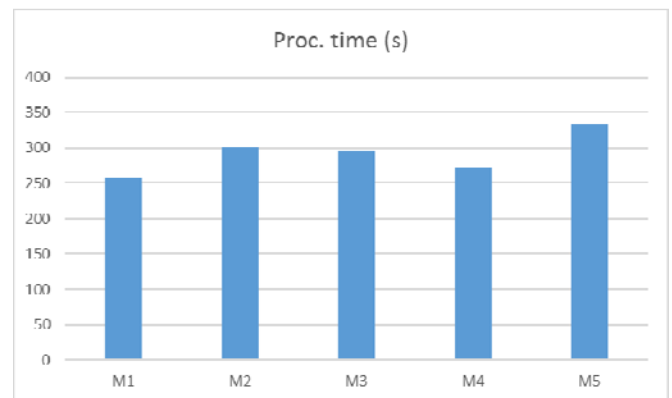
the processing time, however, it provides robust architecture and can cover all data changes during the object lifecycle.

Comparing the results of the model *M4* ensuring version management by background processes with other solutions, slowdown of the model *M3* is 5,82% for the costs, 9,26% for the CPU and 8,56% for processing time (reference model is *M4* – 100%). Model *M2* reached the following slowdown (reference model is *M4* – 100%): 9,50% for the costs, 11,11% for the CPU and 10,17% for the processing time.

TABLE I. RESULTS

	M1	M2	M3	M4	M5
Costs	17 011	19 990	19 232	18 174	22 110
CPU (%)	52	60	59	53	67
Proc. time (s)	257,5	301,1	296,7	273,3	334,8
Size (%)	100	106	106	106	109

Query processing time is the main performance limitation of the whole system. Results are expressed in the fig. 11.



VIII. CONCLUSIONS

Conventional database systems are based on storing only current valid data. Historical images are not the goal of the management and are not reflected. Individual data corrections are versions are targeted to deal with only the most recent ones, as well. Temporal evolution has brought the possibility to cover all the states of the object in the time spectrum. Thanks to that, image of the object or the whole ecosystem at defined time point or interval can be reached. Principle of the uni-temporality is based on the term validity, thus each data tuple is time bordered. In this paper, we extend the paradigm of the temporal database approaches by adding sophisticated module for dealing with data versions. Thanks to that, data can be evaluated anytime with the reflection to the database image used as the input. Each data version is secured by the unique signature hash delimiting data image. Theoretical part of the paper deals with the temporal architectures with emphasis on processed granularity and antiding problem. Own proposed solution architecture is based on version management using

several models, which are experimentally compared. Most important parameters are just used resources and processing time. The best solution is based on the background process extension, by which data versions are covered.

Based on the used environment, proposed solution requires less than 7 percent addition for the processing time and CPU. On the other hand, proposed solution significantly improves performance of the bi-temporal architecture, which deals with the validity and transaction time for versioning. Proposed solution lowers the costs up to 18 percent. Processing time saving is more than 18 percent, as well.

During the future research, we will extend the solution to cover distributed environment complexly. We will deal with automation of the version and error detection in ad-hoc networks to ensure processed data to be always reliable. Solution could be used in any field. One of the strongest sphere is intelligent transport and GPS navigation systems, where particular node can obtain either raw data, if the communication channel is fast, or pre-processed package with emphasis on the security aspect.

ACKNOWLEDGMENT

This publication is the result of the project implementation: *Centre of excellence for systems and services of intelligent transport II.*, ITMS 26220120050 supported by the Research & Development Operational Programme funded by the ERDF.

This paper is also supported by the following project: *"Creating a new diagnostic algorithm for selected cancers,"* ITMS project code: 26220220022 co-financed by the EU and the European Regional Development Fund.



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

REFERENCES

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

- [2] L. Ashdown. T. Kyte "Oracle database concepts", Oracle Press, 2015.
- [3] G. Avilés et al. "Spatio-temporal modeling of financial maps from a joint multidimensional scaling-geostatistical perspective", 2016. In Expert Systems with Applications. Vol. 60, pp. 280-293.
- [4] R. Behling et al., "Derivation of long-term spatiotemporal landslide activity – a multisensor time species approach", 2016. In Remote Sensing of Environment, Vol. 136, pp. 88-104.
- [5] C. J. Date, N. Lorentzos, H. Darwen. "Time and Relational Theory : Temporal Databases in the Relational Model and SQL", Morgan Kaufmann, 2015.
- [6] M. Doroudian, et al: "Multilayered database intrusion detection system for detecting malicious behaviours in big data transaction" IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), 2016
- [7] M. Erlandsson et al., "Spatial and temporal variations of base cation release from chemical weathering a hisscope scale". 2016. In Chemical Geology, Vol. 441, pp. 1-13
- [8] J. Jánošíková, P. Jankovič, M. Kvet, "Improving Emergency System Using Simulation and Optimization", In SOR 17: Proceedings of the 14th International Symposium on Operational Research, 2017, ISBN 978-961-6165-50-1, pp. 269-274
- [9] T. Johnston. "Bi-temporal data – Theory and Practice", Morgan Kaufmann, 2014.
- [10] T. Johnston and R. Weis, "Managing Time in Relational Databases", Morgan Kaufmann, 2010.
- [11] M. Kvassay, E. Zaitseva, J. Kostolny, and V. Levashenko, "Importance analysis of multi-state systems based on integrated direct partial logic derivatives", In 2015 International Conference on Information and Digital Technologies, 2015, pp. 183–195.
- [12] M. Kvet, J. Janáček, "Fair emergency system design under uncertaintyL. In Central European Journal of Operations Research, ISSN 1435-246X, Vol. 26, no. 3, 2018, pp. 599-609
- [13] M. Kvet, K. Matiaško, "Temporal Data Group Management", IEEE conference IDT 2017, 5.7. – 7.7.2017, pp. 218-226
- [14] M. Kvet, K. Matiaško, "Transaction Management in Temporal System", 2014. IEEE conference CISTI 2014, 18.6. – 21.6.2014, pp. 868-873
- [15] M. Kvet and K. Matiaško, "Uni-temporal modelling extension at the object vs. attribute level", IEEE conference UKSim, 20.11 – 22. 11.2014, , pp. 6-11, 2013.
- [16] D. Kuhn, S. Alapati, B. Padfield, "Expert Oracle Indexing Access Paths", Apress, 2016.
- [17] S. Li, Z. Qin, H. Song. "A Temporal-Spatial Method for Group Detection, Locating and Tracking", In IEEE Access, volume 4, 2016.
- [18] Y. Li et al., "Spatial and temporal distribution of novel species in China", 2016. In Chinese Journal of Ecology, Vol. 35, No. 7, pp. 1684-1690.