Journal of Information Systems Engineering & Management, 2018, 3(2), 13
ISSN: 2468-4376

Temporal Data Performance Optimization using Preprocessing Layer

Michal Kvet [1]*, Karol Matiasko [1]

[1] *University of Zilina, Faculty of Management Science and Informatics,* Žilina, SLOVAKIA

***Corresponding Author:** Michal.Kvet@fri.uniza.sk

Citation: Kvet, M. and Matiasko, K. (2018). Temporal Data Performance Optimization using Preprocessing Layer. *Journal of Information Systems Engineering & Management*, 3(2), 13. https://doi.org/10.20897/jisem.201813

Published: April 06, 2018

ABSTRACT

Temporal databases forming intelligent information systems store wide range of data structures with significant data amount evolved over the time. In this paper, history of temporal models is described with regards on their usage a limitations. Performance is important part and main technology characteristics and must be handled with emphasis. Therefore, we propose two layers extending the optimizer evaluations by transforming the original user query. Preprocessing layer deals with undefined states as well as condition management of the query, which is transformed based on defined index structures and data dictionary statistics. The aim is to improve performance highlighting conversions and reducing condition evaluations. Therefore, index access paths are used, which are reflected also in the Experiment section of this paper.

Keywords: temporal database, validity, undefined state, MaxValueTime, UndefTypeValue, index, optimizer preprocessing

INTRODUCTION

Intelligent information systems are supported by complex decision making, data processing and processes. The core part for the evaluation and complexity forms the data management, which must deal with more and more data amount over the time. Database systems are the resources for data management, preprocessing, evaluating and handling (Janáček and Kvet, 2015; Rocha and Lima, 2018). Storage capacities are now widespread. Thus, nowadays, we are more or less able to store anything regardless the structure or size. However, significant factor is just the effectivity of such system. Data characteristics evolve over the time and therefore, it is necessary to monitor the evolution, evaluate the significance and impacts of changes with emphasis on future prognoses and reactions to the situation formed in the future, but based on actual conditions. Temporal databases offer ideal solution for data management over the time. In the first part of the paper, historical evolution of temporality is mentioned with regards on the temporal characteristics, goals and means of data storage. As we can see, multiple development streams can be felt, but most of them are, unfortunately, negatively influenced by the impossibility of adopting new temporal standard. It means, as the consequence, that the temporal development and research in this field has been reduced over the years. Thus, approved standard of the relational database approach is still based on conventional principles, which means, that only actual valid data are stored effectively and each new state characteristics replace the existing data tuple. Such approach is really effective, if no historical data, nor future valid data have to been stored. Performance of the database system as the core part of the information system is reflected by multiple steps and fields. It reflects the database architecture, administration properties and settings, query optimization and index structures. In this paper, we deal with the processing steps of the query evaluations having initially request data operations till the point of obtaining result sets. Several steps had to be done by optimizer and support background processes to get the right solution for data access. In our research, we extend the principles by adding new layer for the pre-processing and optional post- processing techniques to get reliable data in required form as soon as possible. As we can see, our approach fills the gap of the processing and prepares better environment for optimizer based on actual index structures and transcript of the original query to the new one, which will in the last phase contain the same data, however, provided more effectively. For the purposes of this paper, we will describe *NULL* value management problems, implicit conversions and conditions handling of the query.

HISTORICAL TEMPORAL CHARACTERISTICS EVOLUTION

Development principles, characteristics, approaches and methods evolve over the time, are improved or even replaced by newer ones. Complex data are stored in the databases covering current trends. Common point grouping data together is just time. The requirement for accessing historical data was created almost in the same time as databases themselves. Researchers perceived the strength and consequences of historical data management, however, in the first phase, it was strictly limited by the hardware resources – disc storage capacity and price of the whole system in comparison with data profit. Larger volumes and price reductions has brought the opportunities for temporal extensions, firstly managed by the backups and log files, where, almost now, all historical data can be obtained. Effectivity is, however, very poor – log files contain all transactional data regardless the temporal attribute changes, backups as well as log files are usually stored only during strictly defined time period, afterwards, they are deleted due to disc space requirements. Thus, historical data could be obtained during the short time. Later, at the end of 80s of 20th century, architectural temporal model based on historical tables was proposed (Ahsan and Vijay, 2014; Arora, 2015). Historical data were separated from actual state conditions. It provides significantly better solution based on effectivity of data management and disc space demands. Thus, wider time spectrum was able to be managed. Even with further price reductions of the hardware components, solution became more and more expanded. As a consequence, future valid approach has been proposed with the same characteristics and historical tables – data model was divided into three parts – historical data, current data and future valid data. However, the management of the states was manual resulting in inconsistencies, update delays and loss of solution reliability. Therefore, during the 90ties of 20 century, object level temporal model has been developed consisting of two temporal spectrum – logical and physical time. Logical time corresponds the *validity* called by computer science and reflected to *effective* time notation referred by IT professionals. Validity of the object itself requires specific and non-overlapping time period – an authorized description of the object state during particular period of time. Physical time expresses the time of tuple creation, usually modelled by single date. To ensure consistency and integrity of the temporal database, new paradigm needed to be defined (Arora, 2015; Gunturi and Shekhar, 2017; Chomicki and Wihsen, 2016). Soon, extension of the SQL statements highlighting temporal layer, was proposed, however it has been never approved as standard resulting in development abolition in this field. Later, the attribute oriented approach have been proposed. All previous solutions were based on object modelling itself, by extension of the primary key using temporal characteristics. It has, however, significant disadvantages. If some data portions do not evolve over the time (static attributes) or its evolution is not necessary to be stored (conventional attributes) or even cannot be stored due to security reasons, existing approaches are not suitable. Attribute oriented approach extends the principles and models temporality using attribute granularity using three layer architecture. The principles and structure are described in (Kvet and Matiaško, 2017a; Kvet and Matiaško, 2017b). In 2016, modelling extension has been proposed, which uses grouping factors. It can be considered as the hybrid solution between object level and attribute oriented approach. Namely, some data portions can be synchronized and provided in specific form in strictly defined time. Attribute oriented approach would require to store each attribute separately, multiple *Insert* statements would be generated and processed. Therefore, groups of attributes are created, which can then processed as one attribute reflecting only one *Insert* statement into temporal system. In this case, however, it is inevitable to mention, that these groups are also temporal, their characteristics can evolve over the time and should be envisaged. Such solution can consists also of spatial data forming spatio-temporal solution. Principles and other temporal solutions are described in (Claramunt et al., 2015; Goevert et al., 2016; Johnston, 2014). Special case covers Big Data and Real Time systems (Sun and Huang, 2016).

QUERY OPTIMIZATION

There are several layers and interfaces between users and database itself. Due to security reasons, user cannot have direct access to database, which guarantees consistency and correctness of the stored data, managed also by other background processes. User is in the architecture represented by the *user process* on the client site. Communication is based on the server process created by background process *Process Monitor* (PMON) started by

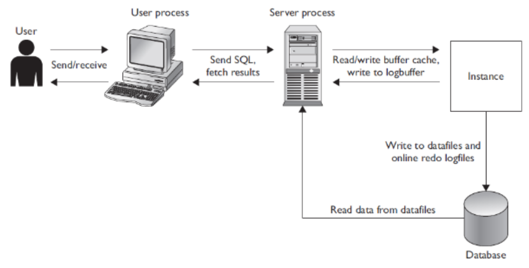


Figure 1. Query processing stages

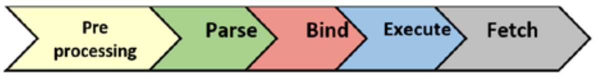


Figure 2. Proposed query processing stage extension

listener. The *server process* is a process running on the database server machine that executes the SQL (processing stages are shown in **Figure 1**). Receiving and processing user is crucial part of the client-server architecture.

Existing database system defines the execution of the SQL query through four stages:

- *parse*, *bind*, *execute* and *fetch*.

First of all, system checks, whether the query is not already parsed and execution plan prepared. If so, existing stored version in the *library cache* is used with defined execution plan. Aim of the *parsing* phase is to get the path of best possible execution of the query. It works out, what the statement actually means. It interacts with instance *shared pool*, which is used to convert SQL into executable form. In the *bind* phase, variables are expanded into literals. Afterwards, *execution* phase is started. During this phase, data are read from the *buffer cache*, if applicable. Such data can be, however, too new in comparison with the timepoint of the query evaluation start point. Therefore, to solve such problems, *log buffer data* and *log files* are used to get the consistent image of the particular data blocks based on the same *change number value* (*SCN*). If the consistent image is not possible to build, e.g. due to rewriting the logs, the following exception would be raised: *ORA-01555 Snapshot Too Old*. If the relevant blocks are not located in the memory buffer cache, data files using *tablespace*s must be located and particular blocks copied to the *buffer cache*. Then, the evaluation is the same. Finally, result set in created and offered to the user reflecting the fetch phase. it is transferred to the user process.

For our processing and evaluations, *parse* phase is the most significant and used for another optimization to get relevant data sooner. Data treatment is in fact dealing with access methods, using defined indexes according to the actual statistics (**Figure 2**). Our approach extends the stages by preprocessing as the first phase.

INDEXES & ACCESS METHODS

One of the main features of optimization is based on using index structures. Temporal databases are oriented for state management and monitoring over the time. Getting states and individual changes in the *Select* statement forms the core of a major milestone of efficiency and speed of the system (Johnston, 2014; Johnston and Weis, 2010).

Index is defined as an optional structure associated with a table or table cluster that can sometimes speed data access. By creating an index on one or more columns of a table, you gain the ability in some cases to retrieve a small set of randomly distributed rows from the table. Indexes are one of many means of reducing disk I/O. If a heap-organized table has no indexes, then the database must perform a full table scan to find a value.

The absence or presence of an index does not require a change in the wording of any SQL statement. An index is a fast access path to a single row of data. It affects only the speed of execution. Given a data value that has been indexed, the index points directly to the location of the rows containing that value. Database management system automatically maintains the created indexes – changes (*Insert, Delete, Update*) are automatically reflected into index structures. However, the presence of many indexes on a table degrades the performance because the database must also update the indexes.

The index structure of the B+tree is mostly used because it maintains the efficiency despite frequent changes of records. B+tree index consists of a balanced tree in which each path from the root to the leaf has the same length. In this structure, we distinguish three types of nodes - root, internal node and leaf node. B+tree extends the concept of B-tree by chaining nodes at leaf level, which allows faster data sorting. DBS Oracle (used for experiments) defines the accessibility methods of two-way linked list, which makes it possible to sort ascending and descending, too (Johnston and Weis, 2010; Kuhn, 2012).

There are several types of access methods influencing the performance and method of obtaining the required data. Usage of a particular access method depends on several factors, firstly defining the suitability of the defined index and amount of data that is obtained in comparison with total amount of data in particular table. Suitability of the index is mostly based on the order of indexed columns, respectively the presence of them in the index.

In this case, we can define multiple scenarios to explain individual access methods:

1. No suitable index is defined. In that case, it is necessary to scan all the data blocks associated with the table sequentially, therefore *table access full method* is used with regards on High Water Mark (Johnston and Weis, 2010; Kuhn, 2012).
2. There is no condition in the query to limit data amount. Despite the fact, that we can use index, it is not appropriate because we have to access data blocks, so there will be one additional step of the processing without significant improvement. Vice versa, using such index will usually cause even processing delays and rising requirements and sources. *Table access full* method is used, too.
3. All required data are in the index. This case defines two situations – the order of the indexed attributes is suitable starting with *Where* clause followed by the *Select* clause. In this situation *Index range scan* for getting data range is used, if only one value (row) should be provided, *Index unique scan* is treated. Another situation occurs, if all required data are in the index, but the conditions in the *Where* clause do not allow to use index by trimming some branches of the processing away. Using index is suitable and faster in comparison with whole data block accessing, however, the whole index structure has to be processed. Thus, *index fast full scan* is used.

There are also another index types like bitmaps, hashes, functional and virtual indexes, which are described in (Arora, 2015; Kuhn, 2012; Sun and Huang, 2016).

PROPOSED PREPROCESSING LAYER

Creating effective solution with emphasis on performance, we extend the existing sequence of the optimizer processing to get more reliable data and use more effective index approach resulting in better performance – reducing system resources, costs of the processing and also time.

We will describe three principles, which will improve performance. All these solutions are developed, experienced and experimented. Query transformation as well as layer for the post-processing ensuring correct data outputs is done automatically, if the corrector is launched on the server. It requires less than 0,5% of the processing resources, however the improvements are far better.

First part of the preprocessing layer consists of the undefined values management. It must be emphasized, that such activities are usually modelled using *NULL* values, however, it can have special attribute denotation in the temporal system. Moreover, it has significant performance limitations – *NULL* values are not indexed using B-tree index structures at all. As a consequence, *Table Access Full* is used. *Bitmap* index data structure as the second index type based on usage quantity, does not provide sufficient power, although it can manage *NULL* values, because it main limitation is just low cardinality of column values, however, for date as well as sensor data processing, such approach is completely unsuitable. *Bitmap* index is mostly developed and commonly used for data warehouses and decision support systems. Moreover, it significantly degrades performance, when multiple *Update* statements are used. It is necessary to emphasize, that temporal data are mainly characterized by strong update streams.

Solution is based on *MaxValueTime* and *UndefTypeValue* notation, which reflects the replacing of the undefined value using pointer to the undefined value object stored in the data dictionary. It has been proposed as standalone definition, which must be user managed explicitly. Here, it is part of the automated management techniques. Therefore, we can divide *NULL* values themselves and undefined values on the other sphere.

The principle is based on array definition consisting of all used data types, which should be covered by undefined management notation. Such object is stored in the data dictionary and invoked into the memory during the startup, respectively opening process of the database. Then, it is permanently stored in the memory, until the database is closed – in this case, the prospective changes on these objects (registering new data types, removing unusable) are reflected and stored in the database. Each data type has assigned unique identifier consisting of the 1 byte size, which is then used as undefined value pointer.

Figure 3 shows the architecture of the proposed undefined value management solution. Left part of the figure represents the object with its state evolution over the time. For the simplicity, time spectrum is replaced by the sequence number. Such object is stored physically in the database (data files in the disc space) and optionally in the memory *buffer cache* during the processing. Pointer locates data dictionary object always located in the memory, if the database instance is open. Right part of the figure represents data dictionary object highlighting the undefinition of the values. Yellow numbered parts reflects individual states of the objects, specifical attribute values are modelled inside. If undefined value is used, pointer to data dictionary object is used (in the right part). Each

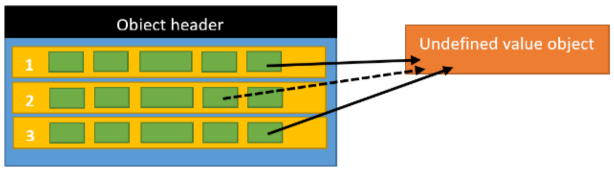


Figure 3. Undefined value management architecture

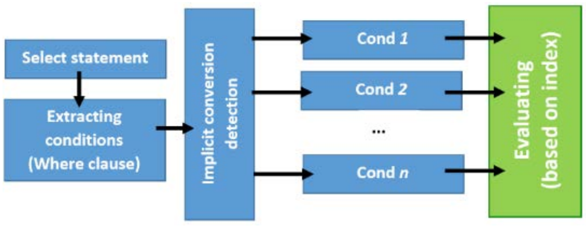

Select statement
Extracting conditions (Where clause)
Implicit conversion detection
Cond 1
Cond 2
...
Cond n
Evaluating (based on index)

Figure 4. Implicit conversion detection

attribute data type has its own submodel of the object (reflected by the another line type). Arrows in the Figure 3 model the pointers.

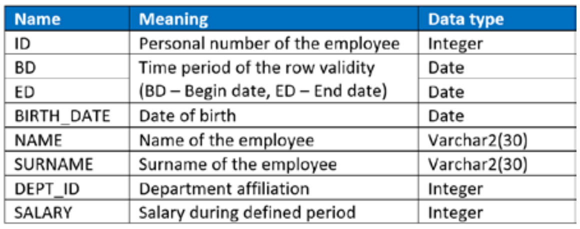
The second part forming the preprocessing layer, is based on implicit conversions. As we know, data can be automatically transformed to another data type and compared in the *Where* clause of the *Select* statement. However, what about the performance? There is a significant performance issue based on data comparison. B+tree index store particular values in the leaf layer sorted, so the range comparison is really fast. However, when the data are transformed (e.g. integer values are transformed into strings), sorting principle is corrupted. Therefore the *Index Range Scan* method is shifted to *Full Index Scan*, respectively *Fast Full Index Scan*, which means, that data are handled in the index in unsorted way, thus the performance is worse in comparison with standard *Range Scan* access (Kuhn, 2012). Automatic conversion management in the preprocessing layer is based on defined index, function based index and the whole processing approaches. Automatic evaluation reflects the defined indexes.

Figure 4 shows the implementation principle. Preprocessing layer extracts the *Where* clause conditions to the separate ones, implicit conversions are identified and replaced based on defined index. If no suitable is defined, original condition is left alone and processing is shifted for server management.

Defined conditions of the query often do not cover index definition, thus *Table Access Full* must be used regardless the number of the data stored in the table and reduction factor expressing the ratio between all table data and cardinality of the result set. The third part of the preprocessing layer deals with condition evaluations. Previous mentioned module divides conditions to the separate ones. After the implicit conversion evaluations, another process is launched. The aim is to distinguish conditions, which can improve processing method and performance itself. When original command is not suitable for index processing, in principles, two cases are possible – all data blocks must be processed sequentially. However, we propose another solution. It is based on condition transformation – some can be added, some can be even removed. However, it is only the first layer of the processing, thus user defining the query cannot get incorrect data. Important part of the condition evaluation is, that the result set of any processing step cannot be smaller than result set of the original query. Due to, *Table Access Full* method is obviously transformed into *Index Scans*, optimal solution provides index, which does not require accessing physical data blocks in the disc at all, however such option is rather occasional. Therefore missing data are loaded from the disc forming the first level of the processing. Preprocessing phase as well as parse, bind and execute phase have been executed. Now, result set is provided, however, it contains larger data amount in comparison with original query. In this step, therefore, important role plays post processing layer.

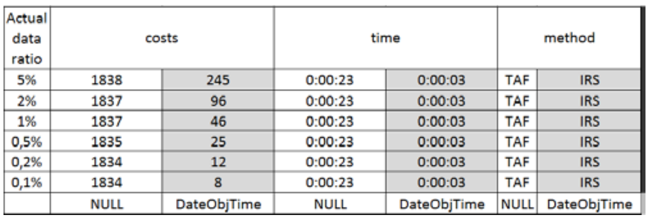
POSTPROCESSING LAYER

Post processing is a crucial part of the processing, if some conditions are corrected, replaced or even removed to highlight the performance. Input to this layer is the result set obtained by evaluating modified query. In this step, obtained data must be filtered based on original conditions. Each data tuple is evaluated and if it fits the original conditions, it is moved to the real output set, which will be later send to the user as response to the defined query. How it works, that performance of the model extended by the pre-processing and post-processing layer is better than accessing all data blocks? The principle resides in three reduction factor types, which are initially evaluated and then, module decides, whether adding or removing conditions can significantly influence performance or not. In our development, performance improvement must be higher than 10% in comparison with original conventional approach defined by optimizer that this method can even be running. It ensures, that the it will provide positive effect in terms of performance and processing costs.



Name	Meaning	Data type
ID	Personal number of the employee	Integer
BD	Time period of the row validity	Date
ED	(BD – Begin date, ED – End date)	Date
BIRTH_DATE	Date of birth	Date
NAME	Name of the employee	Varchar2(30)
SURNAME	Surname of the employee	Varchar2(30)
DEPT_ID	Department affiliation	Integer
SALARY	Salary during defined period	Integer

Figure 5. Experiment table structure



Actual data ratio	costs		time		method	
5%	1838	245	0:00:23	0:00:03	TAF	IRS
2%	1837	96	0:00:23	0:00:03	TAF	IRS
1%	1837	46	0:00:23	0:00:03	TAF	IRS
0,5%	1835	25	0:00:23	0:00:03	TAF	IRS
0,2%	1834	12	0:00:23	0:00:03	TAF	IRS
0,1%	1834	8	0:00:23	0:00:03	TAF	IRS
	NULL	DateObjTime	NULL	DateObjTime	NULL	DateObjTime

Figure 6. Performance of the undefined time value management

At first glance it may seem that this method requires more system resources than original ones. That is absolutely true, but only if the optimization system resources are compared. However, when dealing with complex query definition, evaluation and processing, data amount is decreased significantly resulting in positive processing costs and time – not the whole data table is processed, just the part of the table delimited by defined conditions. Thus, the aim is to minimize the difference between original data amount and query executed by our processing layer.

PERFORMANCE

Our experiments and evaluations were performed using defined example table - employee. **Figure 5** shows the structure of the table. 50 departments were used, each consisting of 1000 employees, each of them was delimited by 20 different salaries over the time. Thus, total number of rows was one million. No primary key was defined, because of the environment properties and our direct opportunity for explicit index definition. Column level temporal architecture has been used.

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 and HDD: 500GB.

Experiment results comparison was obtained using *autotracing*. These parameters were monitored:

- *access method (operation),*
- *CPU costs (in [%]),*
- *processing time (in [hh:mi:ss]).*

Data export can be downloaded in *mknet.fri.uniza.sk/Shared/Employees.zip*.

Undefined values management have been reflected by the atribute *ED* (end date of the validity), which is in standard environment undefined during the *Insert* operation of the actual data.

Whereas temporal characteristic requires each state to be defined by no more than one row, our defined environment limits the number of actual states to 50 000. In the following experiments, various number of actual states is used – 0,1%, 0,2%, 0,5%, 1%, 2%, 5% (Kvet and Matiaško, 2017).

Comparison of undefined time value denoted by *NULL* value in comparison with *DateObjTime* database object solution (our developed model) is used. B+tree index based on attributes *ED*, *BD* and *ID* is created (in such defined order) – the reason and comparison of index types can be found in (Johnston, 2014). *Select* clause of the statement consists of only *ID* attribute, thus, all data can be obtained using index with no necessity for accessing data files in the physical storage. **Figure 6** shows the experiment results. As we can see, if *NULL* values are used, there is no other way, so *Table Access Full (TAF)* method must be used to avoid *NULLs*. If undefined value is modelled by our *DateObjTime* solution, all values are indexed, so *Index Range Scan (IRS)* with significant performance improvement can be used. Total costs and processing time reflect significant performance growth, too. If all data have actual non-limited value, 86, 67% of costs is eliminated, which reflects 86, 96% of processing time. With the reduction of the number of undefined values, the difference is even more strict – 99, 56% of costs and 86, 96%. As we can see, processing time does not depend on number of actual data ratio. The reason is based on necessity of index loading into memory, which reflects the same time.

Graphical representation of the solution is in **Figure 7**.

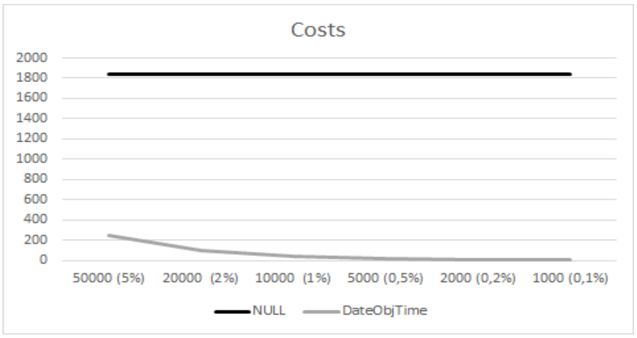
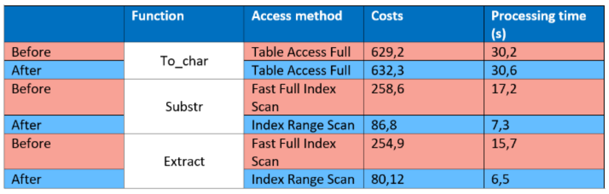


Figure 7. Processing costs



	Function	Access method	Costs	Processing time (s)
Before	To_char	Table Access Full	629,2	30,2
After		Table Access Full	632,3	30,6
Before	Substr	Fast Full Index Scan	258,6	17,2
After		Index Range Scan	86,8	7,3
Before	Extract	Fast Full Index Scan	254,9	15,7
After		Index Range Scan	80,12	6,5

Figure 8. Conversion method performance

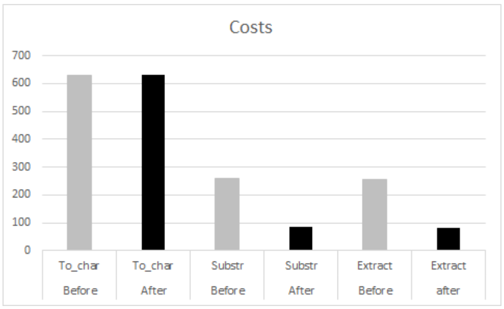


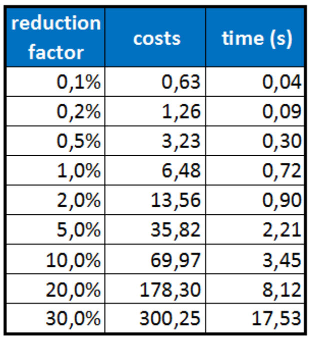
Figure 9. Conversion method performance

Second part of the experiment section deals with the implicit conversions and function results processing. The principles will be described using the *birth_date* attribute. European *NLS_DATE_FORMAT* has been used (*DD.MM.YYYY*). To get the year of the birth, several solutions can be used with various data types providing. The first function category used covers *Extract* and *To_char* function, which can be used regardless the *NLS_DATE_FORMAT* parameter settings. The second group covers *substr* function. In this case, date is automatically converted to string. As we can see, results are significantly different. Our solution manages and rewrites conditions based on actual server settings and indexes automatically. **Figure 8** and **Figure 9** shows the results. As we can see, implicit data conversion management can bring wide range of problems resulting in using inappropriate access method chosen bz the optimizer. *To_char* method is immune, whereas in all cases, all table blocks are scanned. However, when dealing with *substr* method, based on defined index, performance can be improved using *66,43%* for costs and *43.79%* for processing time. It is based on shifting access method from *Fast Full Index Scan* to *Index Range Scan.*

Extract method is not influenced by the date format session parameter and provides also significant performance improvements – *68,57%* for costs and *58,60%* for processing time. As we can see, all these methods would provide the same results, however, our proposed solution automatically evaluates and replaces them to get better performance and reduce system resources.

The last characteristics are based on conditions management using preprocessing and postprocessing layer. In this case, reduction factor study has been performed. Using the statistics stored in the data dictionary, input rating of accessing all data blocks in comparison with using defined index (original conditions are limited) is evaluated. Then, also output rating is calculated. It reflects the ratio between processed data using added or removed conditions and the original query result itself. It must be emphasized, that the size of each step is counted and evaluated based on statistics, therefore it is inevitable to retain statistics actual. To prevent incorrect decision making, if the statistics are older than 2 days (by default), condition evaluation module is not launched and original solution is used. Results based on reduction factor are shown in **Figure 10**. Graphical representation is in **Figure 11**.

Reduction factor defines the upper limit of the amount of data to be processed compared to the total amount. As we can see, even for 30% data to be processed, performance reflects the improvement using 52,27% (full access is reference model).



reduction factor	costs	time (s)
0,1%	0,63	0,04
0,2%	1,26	0,09
0,5%	3,23	0,30
1,0%	6,48	0,72
2,0%	13,56	0,90
5,0%	35,82	2,21
10,0%	69,97	3,45
20,0%	178,30	8,12
30,0%	300,25	17,53

Figure 10. Reduction factor and performance

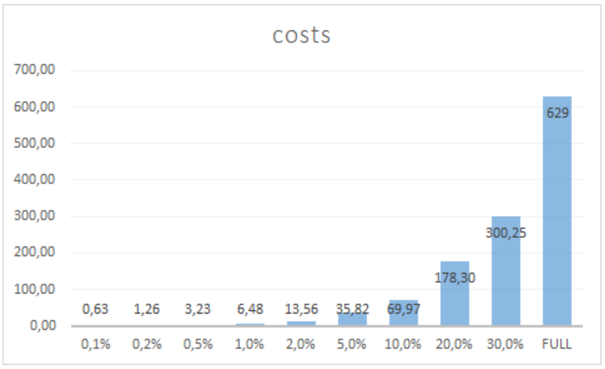


Figure 11. Conversion method performance

CONCLUSIONS

Intelligence of the current information system is based on complex data management with regards on historical and also future valid object states. Therefore temporal database management can be considered as core of such systems. Whereas the amount of data to be processed, stored and evaluated, is still rising, it is necessary to manage, store, process and evaluate them effectively. Index definition is only one part of the complexity, because it is not possible to define index for each query, whereas the query type and also evolve. Moreover, too many indexes can very negatively influence performance, such index tree must be rebalanced during the *Insert*, *Update* and *Delete* operation causing processing delays.

Our solution proposes extension of the query processing using preprocessing layer, which can extend the technology of the query evaluation using undefined value management directly in the index and condition management. Condition interpretation is divided into two parts – handling implicit conversions and condition reduction. Therefore, defined index can be used. Consequently, original conditions are applied into preprocessed result set, which significantly shortens the data collections to be processed. Thus, performance is optimized.

Proposed solution is always launched automatically, if permitted. No other settings is necessary to be set up, however, as we can see in the Experiment section, there is significant performance benefit. In the future, we would like to extend the solution to the distributed environment, in which each node stores and proposes defined index fragments. The aim is to optimize query based on offered capabilities of the particular node, respectively to associate query to the optimal evaluation node place.

REFERENCES

Ahsan, K. and Vijay, P. (2014). *Temporal Databases: Information Systems*. Booktango.

Arora, S. (2015). A comparative study on temporal database models: A survey. *2015 International Symposium on Advanced Computing and Communication* (ISACC), 14-15 Sept. 2015, IEEE. https://doi.org/10.1109/ISACC.2015.7377335

Claramunt, Ch., Schneider, M., Wong, R. and Xiong L. (2015). Advances in Spatial and Temporal Databases. *14th International Symposium*, SSTD 2015, Hong Kong, China, August 26-28, 2015. Proceedings (Lecture Notes in Computer Science), Springer. https://doi.org/10.1007/978-3-319-22363-6

Chomicki, J. and Wihsen, J. (2016). Consistent Query Answering for Atemporal Constraints over Temporal Databases. *2016 23rd International Symposium on Temporal Representation and Reasoning* (TIME). 17-19 Oct. 2016. IEEE. https://doi.org/10.1109/TIME.2016.23

Goevert, K., Cloutier, R., Roth, M. and Lindemann, U. (2016). Concept of system architecture database analysis. *IEEE International Conference on Industrial Engineering and Engineering Management* (IEEM), 4-7 Dec. 2016, IEEE. https://doi.org/10.1109/IEEM.2016.7797907

Gunturi, M. V. and Shekhar, S. (2017). *Spatio-Temporal Graph Data Analytics*. Springer. https://doi.org/10.1007/978-3-319-67771-2

Janáček, J. and Kvet, M. (2015). Min-Max Optimization of Emergency Service System By Exposing Constraints. *Communications: Scientific Letters of the University of Žilina*. Volume 2/2015, pp. 15 – 22, EDIS.

Johnston, T. (2014). *Bi-temporal data – Theory and Practice*. Morgan Kaufmann.

Johnston, J. and Weis, R. (2010). Managing Time in Relational Databases. Morgan Kaufmann.

Kvet, M. and Matiaško, K. (2017). Database optimizer extensions using preprocessing layer in temporal environment. In *Information Systems and Technologies (CISTI), 2017 12th Iberian Conference on* (pp. 1627-1633). IEEE. https://doi.org/10.23919/CISTI.2017.7975682

Kvet, M. and Matiaško, K. (2017). Temporal Transaction Integrity Constraints Management. *Cluster Computing*, Vol. 1/2017, Springer. https://doi.org/10.1007/s10586-017-0740-8

Kuhn, D. (2012). *Expert Indexing in Oracle Database* 11g. Apress.

Rocha, A. and Lima, S. (2018). *Expert systems: The journal of knowledge engineering special issue on WorldCist'16 - 4th world conference on information systems and technologies*. Blackwell Publishing Ltd

Sun, D. and Huang, R. (2016). *A Stable Online Scheduling Strategy for Real-Time Stream Computing Over Fluctuating Big Data Streams*, pp. 8593–8607, IEEE Access. https://doi.org/10.1109/ACCESS.2016.2634557