# Temporal bi-index

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

*Abstract*— **Currently, it is important to store the whole evolution of the object states by referencing the validity time frame. Temporal databases are inseparable parts of intelligent information systems. Temporal spheres are mostly referring validity expressed by the start and end timepoints. To ensure performance, database indexes are created by sorting the validity of the states for each object. However, to monitor the evolution in industry monthly, daily, weekly, or annually, the standard conventional index is not suitable. In this paper, temporal architectures are summarized, pointing to the temporal spheres. It mostly emphasizes indexing, focusing on the proposed bi-index combining bitmap and B+tree structure to the common index. Thanks to that, evolution monitoring for the specified time range can be significantly improved.**

**Keywords— temporal database; performance; indexing; temporal elements; monitoring**

## I. Introduction

Database storage and overall database technology are inseparable parts of the current information systems forming the interlayer between applications and storage perspectives. There are many types used currently. Although relational databases were first introduced in 60ties of the 20th century, they are still very often used nowadays, whereas they require precise structure delimited by constraints and overall integrity. Relational databases are formed by the entities and relationships between them. The data access is based on relational algebra operations. These operations are part of the execution plan of the query, selected by the database optimizer. Generally, the access is done either through the developed indexes or sequential data block scanning is necessary to be performed – if no suitable index is present or the estimated costs using the index are higher than sequential data block scanning [1] [12]. When dealing with the data warehousing and analytics based on the large data set, performance differences can be significant and overall management should always point to the proper index definition and maintenance.

From the physical perspective, a database is formed by the segment [3], which specifies the structure and general conditions. It also points to the data blocks, which are not, however, created separately, instead, the extents are allocated. Each extent is then represented by the fixed size set of the blocks, generally having 8 blocks. Multiple extents are grouped together forming a linked list. Thus, sequential scanning requires block-by-block memory loading, evaluation, and relevant data tuples extraction. The limitation of that method is just the data block fragmentation and dynamic operations, which can even free data blocks. However, such blocks are not specifically marked and must be treated during the data retrieval block evaluation.

In the past, relational databases were focusing on storing current valid states by replacing original values [14]. Thus, there was no history stored in the system. Although change vectors were part of the transaction logs, it was too difficult, but mostly time demanding to obtain states valid in the defined time interval.

During the evolution, the temporal paradigm has been introduced, forced mostly by the industrial environment. It was primarily was based on storing states of the objects, framed by the temporal spheres, validity, transaction time reference, etc. Thanks to that, any state could be identified, and state evolution could be easily monitored and retrieved [16]. The database systems were shifted to the cloud environment allowing to manage, maintain and store huge data sets very efficiently [7] [11], respecting the ability to reconstruct the data. The main advantage of the Oracle Cloud is just the autonomous aspect of the databases, as well as the auto-indexing option [13] [17] [18]. Autonomous data warehouses and data lakes strongly require sophisticated access methods ensuring performance.

The performance of the querying is strongly associated and supported by the database indexes, by which the relevant data row addresses can be identified. Typically, B+tree indexes are used in relational databases. Such a premise has been applied by temporal processing, too. The validity of the data tuple can be part of the index key, sorting the states in a timeline.

The proposed solution is primarily intended for the Oracle database technology, which is a bit specific. Namely, the Date data type consists of the second precision frame and no other granularities are applicable, compared to, for example, MySQL database, which applies Date, Time, and DateTime data types for storing various precisions and granularities. As a result, for database indexing, temporal value is stored directly, limiting the opportunity to extract individual elements from the Date value (year, month, week, day, hour, etc.) consequencing in defining a function-based index, by which individual elements can be indexed or by requesting sequential data scanning. For the conventional index, data are sorted based on validity, not individual elements. On the other hand, a function-based index requires additional disc storage capacity. Moreover, there can be issues related to the integrity of the values, whereas duplicate tuples would be present.

This paper aims to define bi-index, by which individual elements can be extracted to the separate index structure. The architecture is based on using two index types, interconnected by the logical addresses – IndexIDs. By using the proposed index solution, particular temporal values can be sorted based on any temporal element, allowing you to monitor evolution based on any precision, range, or element. As evaluated in the computational study, general sensor monitoring applied in any environment (transport, medicine, industry, etc. ) can significantly benefit, whereas the proposed solution gets new unique method for the relevant data identification.

The structure of the paper is organized as follows: Section 2 deals with the temporal paradigm by summarizing existing temporal models. Section 3 deals with the indexing providing capability for the data retrieval process. Section 4 deals with the Oracle Date storage principles and element management by pointing to the limitations of the function-based indexes. Section 5 deals with the proposed solution, which is consecutively performance evaluated in section 6.

## II. Temporal Paradigm and Models

The first attempts to develop temporal databases are dated to the 70ties of the 20th century, related to the relational database evolution. It was very clear very soon, that conventional databases storing only current valid states are not suitable for the complexity. Sooner or later, individual state evolution would be necessary to be used, modeled, and highlighted. In these terms, the object identifier is critical. The primary key is used as a unique and inseparable set of attributes offering unambiguous state definitions for the table. It is commonly formed by the ID or composite definition can be used, as well [4] [9] [17]. One way or another, the primary key was considered as a milestone for temporal evolution. Namely, in the first phases, the core solution was based on an object identifier, extended by the validity time frame forming a composite primary key. The limitation was related to efficiency, whereas each change automatically requested storing a new state completely. As a result, individual data changes had to be synchronized, otherwise, there would be many duplicates. Moreover, it was difficult to identify real change. Namely, if the value changed, it did not have to be a real change. The important element was just the accuracy of the measurement, measurement errors, and data availability. Therefore, the architecture of the object-oriented approach was cumbersome, inefficient, and too demanding for the real-world management and reliability of the data stored in that structure.

The opposite granularity solution was introduced in 2013 by focusing on individual attributes. Thus, each change was associated with the attributes, not the whole state. To get the image valid at a defined timepoint, the composition of individual attributes was necessary to be done. On the one hand, it brought efficiency from the point of view of data storage but added a deficiency caused by the creation of a complex state over time [14] [16].

The most suitable solution for applying any update operation frequency for individual attributes is formed by the group-level temporal architecture. The synchronization group aims to manage multiple attributes as a single set if the changes are synchronized. Thus, efficiency is ensured from the storage perspective [17]. Moreover, building data image at a defined timepoint is less demanding. The synchronization group is defined by the temporal layer or the data dictionary can be used as a repository. One way or another, each group is temporarily oriented and is composed, rebuilt, or dropped dynamically based on the defined conditions ensuring overall performance [17] [19]. To limit the necessity to compose current states dynamically from individual attributes or groups, the current valid state layer is also stored separately using object granularity. Thanks to that, even existing solutions dealing with the conventional paradigm can be directly mapped without the necessity to rebuild the solution. Note, that current valid states are formed by the views, instead of physical data, so reliability and storage performance is ensured.

The architecture of the group-level temporal model is shown in fig. 1. It consists of a conventional layer storing current valid states, a temporal layer mapping and composing object states, and a layer storing outdated references. Besides, there is a complex structure and processes to identify and manage synchronization groups autonomously, based on the defined rules and parameters.

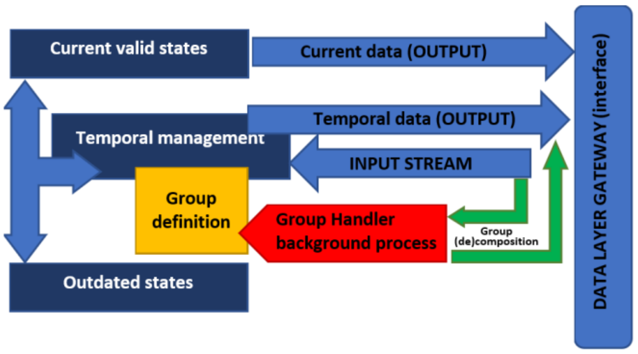


Figure 1. Group-level temporal architecture

Later, the detection and management of synchronization groups were further investigated and optimized.

To compose the database image and monitor changes based on the specified frequency, indexes must be developed, pointing the temporal spheres to individual objects and timeline mapping. Section 3 deals with the index structures and strategies used in temporal systems.

## III. Indexing

The critical part of the information system performance is related to data access. Data retrieval is a staged process of obtaining data and building a result set. Physically, data are stored in the data files, which are block organized. To identify the data, a particular block must be memory loaded and evaluated there. The significant aspect influencing the performance is just the technique to identify relevant data blocks. The best solution can be provided by the index, which takes the address of the row as the output.

In temporal databases, general relational indexes have been primarily inherited. Namely, B+tree is used as a default option, consisting of the root element, internal elements navigating to the leaf layer formed by the key values, and database object reference – ROWID address. ROWID is a physical 10-byte structure, which contains the identifier of the object, the pointer to the particular data file, block, up to the direct position inside the block. Fig. 2 shows the architecture of the B+tree index. For reference and evaluation, selectivity must be treated, expressing the ratio between the amount of unique data portions and total amount. B+tree index prefers many unique values. It is rather wide than deep. Based on [17], the height of the B+tree consisting of 200 millions of keys is only 4.

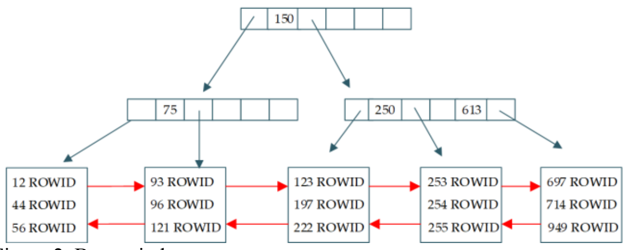


Figure 2. B+tree index structure

The data inside the B+tree are sorted on the leaf layer based on the key and interconnected forming two directional linked lists. Thanks to that, range index scanning can be performed [6] [8] [12].

B+tree is always balanced, which reduces the performance of the change operations. Such a limitation can be partially solved by the post-indexing layer discussed in [15] or Notice lists [16].

The key of the temporal B+tree is the validity time frame or any other temporal spheres, like the timestamp of the insert operation, transaction approval, etc. The limitation of the B+tree index is related to the individual elements. The data cannot be sorted based on the individual elements forming the Date value, like a month, week, or day. Thus, it is really processing time demanding to monitor the changes based on the specified frequency.

The partial solution is related to function-based indexing. The index key is not directly formed by the attribute value, instead, the function result is indexed. The particular solution can be used for individual element extraction. Thanks to that, states valid during the defined start point of the validity are index placed in the same segment, so the scanning is easier, less demanding, but mostly faster. On the other hand, the original temporal value must still be present in the index forming duplicate information inside the index.

Another solution is defined by the index over virtual columns, which are generated dynamically and therefore do not require additional disk space. However, these indexes are not powerful enough for temporal systems.

Bitmap indexes used for analytics and data warehousing require a small number of unique values and a huge amount of data generally. Temporal elements meet this condition and can be used for time-delimited processing. Thus, if the data need to be monitored periodically, particular elements are part of the bitmaps getting the list of referenced ROWIDs. However, individual ROWIDs are not sorted or structured limiting the performance of the monitoring and consecutive decision-making [9] [10]. Fig. 3 shows the bitmap index structure.

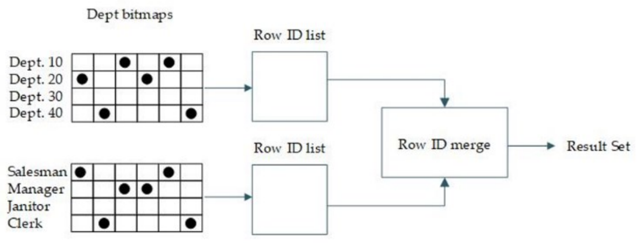


Figure 3. Bitmap index

Section 4 defines the structure of the Date data type used in the Oracle database, compared to the other database systems. It focuses on the individual element coverage, definition, and extraction.

## IV. Oracle Date – Element Management

Oracle Date data type is characterized by 7 bytes in total. The internal data type is 12. It stores date, as well as time values, up to second granularity. There are multiple elements, which can be directly extracted – day, month, year, hour, minute, and seconds and other values can be calculated, like a weekday, day number for the whole year, week reference, etc. All these element values can be extracted using various functions, like TO_CHAR or EXTRACT. All these functions are deterministic, so the output of them can be indexed [5] [6] [20].

The limitation of the Oracle Date data type is just the granularity. If only date values are to be treated, time elements must still be present. Moreover, pure Date value indexing sorts the output based on the whole value and individual elements cannot be treated separately.

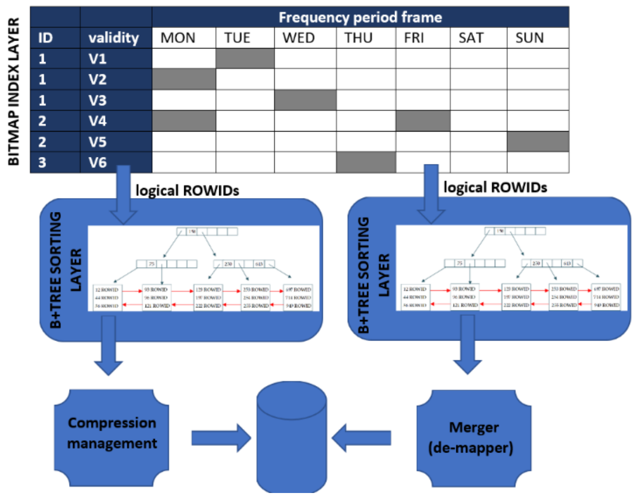
Finer precision can be modeled using a Timestamp data type requiring 11 bytes generally, or 13 bytes by extending the value by the time zone reference (Timestamp with time zone). One way or another, the whole value is always indexed. Therefore, although temporal evolution can be monitored and timeline referenced, getting data periodically reflecting the changes is demanding. The composition of the temporal snapshot image does not get a sufficient solution using existing index types. As evident in the performance evaluation study in section 6, even function-based indexes do not provide a relevant solution for various reasons. Firstly, a composite index would be necessary to be defined with many duplications. Secondly, there is no direct attribute or group reference, thus the object level temporal architecture must be used, or object granularity should be provided by the output, respectively. Thirdly, changing precision and periodicity for the evaluation would require whole index restructuring by dropping the existing one and building the new one. Fourthly, dynamic evaluation across multiple periodicities and dimensions would require multiple indexes to be treated.

Based on these conditions and characteristics, it is inevitable to build a robust solution, which can be applied to any element structure dynamically by ensuring robustness and performance. In section 5, the proposed bi-index is discussed.

## V. Proposed Solution

The limitation of the currently used techniques is the performance if the data states need to be monitored periodically. Individual element extraction must be done by the conversion or extraction functions forcing the developer to create a function-based index. Moreover, if multiple granularities and frequencies need to be monitored, several indexes, which have almost the same structure must be developed. They differ only in the values of the elements themselves and the results of the functions that are indexed. And thus also the order of the element at the leaf level of the B+ tree is different, depending on the processed element. However, the original temporal value must be still present, extended by the row reference using ROWID.

Our proposed solution introduces bi-index, which covers the bitmap index primarily, based on the specific elements, followed by the B+tree processing sorting and locating original data. The solution is based on individual element extraction selectivity, which is low, unlike the core temporal values, which are characterized by high selectivity. These two opposites cannot be part of one index. The architecture of the solution is visually presented in fig. 4. Primarily, individual elements are extracted based on the defined precision and frequency range to be monitored. It is done during the Insert or any change operation, even if the data correction needs to be done. Then, the extracted element value is assigned to the bitmap. Whereas there are low rates of frequencies to be monitored and individual values do not range rapidly, the bitmap index is the optimal solution. E.g. if it needs to be monitored hourly, no more than 24 values are present inside the bitmap map. Similarly, for the minutes or seconds, 60 values are present. When comparing original temporal data provided by the sensorial network, particular pre-processing reaches significant data reduction by grouping them into categories. The output of the bitmap index is the logical reference to the B+tree index. In principle, it can be treated as a local index (for each bitmap member, a separate B+tree is used) or stitched global B+tree index can be used). The main advantage of the local processing and index definition is based on the ability to process data in parallel. The processing and resource demands of the rebalancing are also reduced. As stated, the proposed B+tree forms the second index making the physical data rows accessible through the index by ROWID pointers. Inside the B+tree index, data are grouped based on the extracted element part of the bitmap. This index is used for searching inside the element category, whereas there is a natural assumption, that number of data will be high. Thus, the data are preselected by the bitmap element mapping, followed by the precise row identification using a conventional B+tree index.



| | | Frequency period frame | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ID | validity | MON | TUE | WED | THU | FRI | SAT | SUN |
| 1 | V1 | | ■ | | | | | |
| 1 | V2 | ■ | | | | | | |
| 1 | V3 | | | ■ | | | | |
| 2 | V4 | ■ | | | | ■ | | |
| 2 | V5 | | | | | | | ■ |
| 3 | V6 | | | | ■ | | | |

Figure 4. Bi-index architecture

Compared to table partition development, it allows significantly wider accessibility and flexibility. Namely, partitions can be done based on only one element or the hierarchy can be used, respectively. However, changing the processed granularity and element references is impossible. It would be necessary to reconstruct the whole structure to apply the change. Moreover, without an index definition for the partition, data would be randomly distributed.

## VI. Evaluation Study

Performance evaluation study has been performed in *Oracle Database 21c Enterprise Edition (Release 21.0.0.0.0 – Production)* system environment, characterized by the multi-tenant Real Application Cluster container database. The server parameters were delimited by the following values:

- Processor: AMD Ryzen 5 Pro 5650 2.3GHz
- Operating memory: 64 GB, DDR4 3200MHz,
- Disc storage: 2TB PCIe Gen3 x4 NVMe v1.4, reading 3500 MB/s, writing 3300 MB/s.

The data model was associated with air transport monitoring by pointing to the planned and real routes of the planes. It used a spatio-temporal database. The evaluated table consisted of the attributes defining the flight attributes – identifier of the flight, state sequence number, airspace assignment, data references, and entry and exit time of the particular airspace. The second precision was used, characterized by the Date data type.

It used real flight positions across airspace regions in Europe. For the evaluation, a particular table stored 4 948 094 rows. The data example is shown in fig. 5.

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

Figure 5. Evaluated data structure – Flight information region (FIR) temporal assignment

The computational study focused on the data element pattern identification, coverage, and management. In the first stage, individual flights were categorized by weekdays using European standardization rules (Monday has been marked by value 1, Tuesday referenced value 2, etc.). Thus, in the proposed solution, the weekday was used to form the bitmap consisting of 7 bit-array elements, providing the logical references to B+tree indexes, which referred to the Entry and Exit Time clustering of the data based on flight identification. In the leaf layer, data row addresses were stored.

Among the proposed solution physically modeled either by local (**BI_INDEX_LOCAL**) or global (**BI_INDEX_GLOBAL**) B+index reference, various existing architectures are evaluated to follow the improvement strategy and overall performance. **PURE_B+** solution defines an index managing the temporal data directly without storing individual element extraction separately. It consists of the Entry Time and flight number reference. Then, the index (**FUNC_B+**) is extended by storing the weekday as the master index key element. The third evaluated reference solution points to the weekday treated by the bitmap index (**BITMAP**).

All these solutions, index architectures and access strategies are compared to the environment with no explicit index definition forcing the system to perform sequential data block scanning (**FULL**).

In the evaluation study, firstly, the data access method is evaluated, followed by the processing time demands and total costs. To obtain the data, Autotrace and Explain Plan tools of the SQL Developer were used.

The total processing time to load the data into the structure was 1015.59 seconds.

Monday is the busiest day for air transport, therefore the weekday frequency was evaluated, pointing to the Monday expressed by the element numerically.

The total processing time for the FULL solution 29.09 seconds forcing the system to scan each table-associated block sequentially by loading it into the memory for the evaluation. No data fragmentation, nor empty blocks were present. If this were not the case, the processing time would increase even more, e.g. my moving historical data to the data warehouse, lakes, etc. By using a common B+tree index based on the flight identifier and entry_time (PURE_B+), the total processing time was even 34.54 seconds. The reason was based on forcing the system to use the index. However, the whole index had to be treated, whereas the weekday is not directly present, there and must be extracted. Afterward, the missing attribute values must be loaded from the database using the ROWID pointer. The bitmap index based on the weekday element (BITMAP) calculated during the loading required 19.12 seconds by reducing the number of blocks to be analyzed and evaluated. The list of relevant blocks is obtained by the bitmap index.

By moving the solution to the function-based index (FUNC_B+), processing time was reduced to 16.33 seconds, which is the best solution stated as, grouping existing solutions. However, such a solution is strongly dependent on the frequency and element references to be analyzed. Any change requires a new index to be developed.

The best solution provides our proposed bi-index, which required only 12.06 seconds for the global positioning index (BI_INDEX_GLOBAL). By transforming the weekday evaluation based on the character string, instead of numerical value, processing time demands are 12.52 seconds, reading the 3.86% of additional processing time.

Similarly, if the B+tree index is used locally for each bitmap value, then each particular index stores all the data, so the stitching does not need to be present. Thanks to that, processing time can be generally reduced using 4-5%.

Fig. 6 shows the processing time costs graphically. It is evident, that the bi-index provides the best performance by reducing processing time costs, compared to function-based index using 35.41%. Compared to conventional B+tree, the reduction is 65.08%.

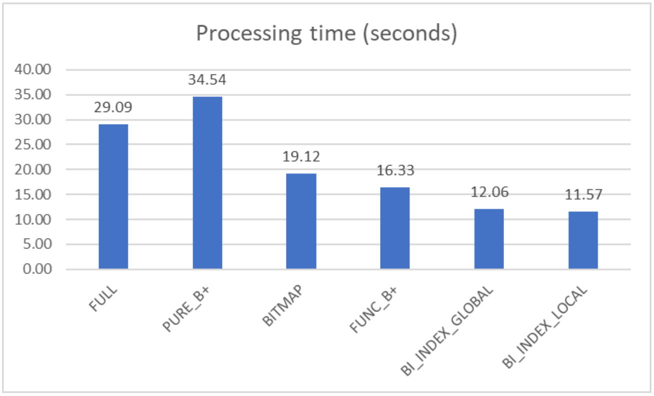


Figure 6. Results - Processing time

Another aspect to be highlighted is the total costs, expressing the technical requirements more complexly, by dealing also with the system sources, I/O operations, waits, consistent image definition, CPU, etc. This value more technically describes individual solutions. Processing time for the sequential scanning is 1778 with no data fragmentations or empty blocks present in the system. By having 10% fragmented data, processing costs would raise to 1991. However, if 50% fragmentation is present, processing costs would be 3325, which expresses an additional 87%. Conventional B+tree and bitmap indexes require almost the same costs, (1197 for PURE_B+ and 1113 for BITMAP) although the structure and storage demands differ significantly. The reason is based on referencing physical infrastructure by the database blocks, which is almost the same. Function-based index extracting particular weekday value reduces the costs to 543, whereas the data are already pre-calculated during the Insert operation. Comparing the proposed bi-index and function-based index, processing time costs are almost the same, too. It results in a structure extension of the bi-index, which requires two indexes processed sequentially. However, dealing with the processing time as a critical parameter for the user, the proposed bi-index reaches significantly better performance. Fig. 7 shows the results graphically.

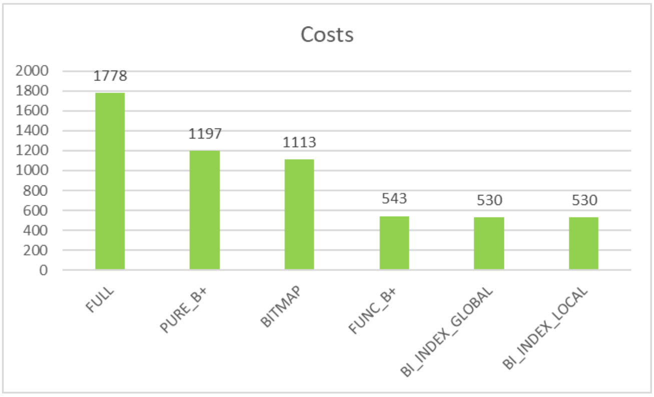


Figure 7. Results - Costs

Finally, the evaluation emphasizes the storage demands, delimited by the index structure size. Graphical representation is shown in fig. 8. It is evident, that the most demanding is a function-based index, whereas it extends the index key by holding a weekday value, which is, compared to the bi-index stored once for each tuple. Bi-index marks it only once in the bitmap. Conventional B+tree requires 108 MB, storing function extraction results, storage demands are 116 MB.

By evaluating textual and numerical values for the bi-index, numerical representation reflects 112 MB, while textual denotation takes 128 MB.

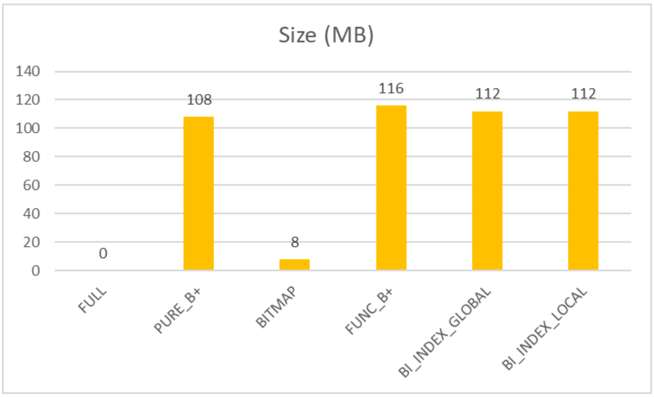


Figure 8. Results - Size

## VII. Conclusions

Current trends in database technology point to the complexity of the processing, by shifting the management from the non-temporal approach to storing the whole state evolution and timeline validity reference. Such a premise requires a sophisticated definition of the temporality, either from the data model point of view, but also from performance aspects should be emphasized reaching robustness and complexity.

Primarily, the performance of the data processing and retrieval is associated with indexing. Generally, B+tree and bitmaps indexes are used in relational databases, and applied in the temporal approaches, as well.

This paper proposed bi-index as a robust data locator of the temporal aspect, applied for the individual elements. Namely, many times, Individual objects and states should be periodically reflected, identified, and evaluated. By using the proposed bi-index solution, processing time demands can be reduced using 26.15%, compared to the original function-based index in B+tree structure.

The bi-index is formed by the bitmap layer reflecting the temporal extraction element, followed by the B+tree layer pointing to the physical repository. The additional total size demands for the index definition reference 3.45% improvement, compared to the function-based index. It is the consequence of the fact, that the bi-index stores the referred element only once, unlike the function-based index, which processes it once for each covered data row tuple.

In future research, we will emphasize the merging operation of covering elements in multiple bitmaps. In the real environment, various granularities, precision frames, and frequency ranges should be evaluated. Therefore, it is assumed, that the proposed bi-index can significantly help by introducing multiple bitmaps pointing to just one general B+tree, global for each bi-index. The solution will use stitching pointers across the bitmaps. Thanks to that, parallel processing across time frames can be present, reducing the processing time demands. On the other hand, the strong demand for synchronization must be present when states change over time dynamically.

## Acknowledgment

This publication was realized with support of Operational Program Integrated Infrastructure 2014 - 2020 of the project: Intelligent operating and processing systems for UAVs, code ITMS 313011V422, co-financed by the European Regional Development Fund.

EUROPEAN UNION
European Regional Development Fund
OP Integrated Infrastructure 2014 – 2020

MINISTRY
OF TRANSPORT
AND CONSTRUCTION
OF THE SLOVAK REPUBLIC

It was partially supported by the Erasmus+ project: Project number: 022-1-SK01-KA220-HED-000089149, Project title: Including EVERyone in GREEN Data Analysis (EVERGREEN).

Co-funded by the Erasmus+ Programme of the European Union

EverGreen DATA Analytics

## References

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

[2] Al-Sanhani, A.H., Hamdan, A., Al-Thaher, A.B., Al-Dahoud, A.: A comparative analysis of data fragmentation in distributed database. ICIT 2017 - 8th International Conference on Information Technology, Proceedings. 724–729 (2017). https://doi.org/10.1109/ICITECH.2017.8079934

[3] Castro-Leon, E., Harmon, R.: Cloud as a Service, Apress, 2016

[4] Dudáš, A., Škrinárová, J, Vesel, E.: Optimization design for parallel coloring of a set of graphs in the high-performance computing. In: Proceedings of 2019 IEEE 15th International Scientific Conference on Informatics, pp 93–99. ISBN 978–1–7281–3178–8

[5] Dunaieva, I., Barbotkina, E., Vecherkov, V., Popovych, V., Pashtetsky, V., Terleev, V., Nikonorov, A., Akimov, L.: Spatial and Temporal Databases For Decision Making and Forecasting. Advances in Intelligent Systems and Computing. 1259 AISC, 198–205 (2019). https://doi.org/10.1007/978-3-030-57453-6_17

[6] Elbahri, F., Al-Sanjary, O., et al.: Difference Comparison of SAP, Oracle, and Microsoft Solutions Based on Cloud ERP Systems: A Review, 15th IEEE International Colloquium on Signal Processing & Its Applications (CSPA), 8-9 March 2019

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

[8] Lew, M.S., Huijsmans, D.P., Denteneer, D.: Optimal keys for image database indexing. In: Del Bimbo, A. (ed.) Image Analysis and Processing. Lecture Notes in Computer Science, vol. 1311, pp. 148–155. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63508-4_117

[9] Mikkilineni, R., Morana, G., Keshan, S.: Demonstration of a New Computing Model to Manage a Distributed Application and Its Resources Using Turing Oracle Design, 25th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 13-15 June 2016

[10] Pendse, S., Krishnaswamy, V., et al.: Oracle Database In-Memory on Active Data Guard: Real-time Analytics on a Standby Database, 2020 IEEE 36th International Conference on Data Engineering (ICDE), 20-24 April 2020

[11] Tanveer, A.: Oracle 19c Data Guard, 2020

[12] Kuhn, D., Alapati, S.R., Padfield, B.: Expert Oracle Indexing and Access Paths. Expert Oracle Indexing and Access Paths. (2016). https://doi.org/10.1007/978-1-4842-1984-3

[13] Kumar, Y., Basha, N., et al.: Oracle High Availability, Disaster Recovery, and Cloud Services: Explore RAC, Data Guard, and Cloud Technology, Apress, 2019

[14] Kvet, M, Matiasko, K., Kvet, M.: Complex time management in databases, Central European Journal of Computer Science vol.4, 2014, pp. 269-284, doi: 10.2479/s13537-014-0207-4

[15] Kvet, M.: Database Block Management using Master Index, FRUCT 32 conference, 9-11 November 2022, Finland

[16] Kvet, M.: Covering Undefined and Untrusted Values by the Database Index, Information Systems and Technologies, Lecture Notes in Networks and Systems, 2022, ISBN: 978-3-031-04828-9

[17] Kvet, M. and Papán, J.: The complexity of the data retrieval process using the proposed index extension, IEEE Access, 2022.

[18] Riaz, A.: Cloud Computing Using Oracle Application Express, Apress, 2019

[19] Rolik, O., Ulianytska, K., Khmeliuk, M., Khmeliuk, V., Kolomiiets, U.: Increase Efficiency of Relational Databases Using Instruments of Second Normal Form. 221–225 (2022). https://doi.org/10.1109/ATIT54053.2021.9678605

[20] Schreiner, W., Steingartner, W., Novitzká, V.: A novel categorical approach to semantics of relational first-order logic. Symmetry 12(1584), 2020 (2020)