IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 27 March 2024, accepted 5 April 2024, date of publication 10 April 2024, date of current version 22 April 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3387046

RESEARCH ARTICLE

Treating Temporal Function References in Relational Database Management System

MICHAL KVET[1], (Member, IEEE), **JOZEF PAPAN**[2], AND **MARTINA HRÍNOVÁ DURNEKOVÁ**[1]

[1]Department of Informatics, University of Žilina, 010 26 Žilina, Slovakia
[2]Department of Information Networks, University of Žilina, 010 26 Žilina, Slovakia

Corresponding author: Michal Kvet (Michal.Kvet@uniza.sk)

This work was supported by the Vedecká grantová agentúra Ministerstva školstva, výskumu, vývoja a mládeže Slovenskej republiky a Slovenskej akadémie vied (VEGA) Project, Developing and Applying Advanced Techniques for Efficient Processing of Large-Scale Data in the Intelligent Transport Systems Environment, under Grant 1/0192/24.

ABSTRACT Databases form an inseparable part of the information systems. They hold the data formed by the data models. Data are operated by the database management system, which takes an interlayer between the data and the external environment. They can be purely obtained from the database using the Select statement or can act as the inputs for the function calls. Data can be temporally or positionally oriented, allowing the storage of multiple tuple states for one object. Temporal databases can be defined by various precision frames and granularities, from object level through synchronization groups up to the est attribute granularity. Inspired by temporal databases, we provide new solutions for managing functions, and code can also evolve over time. Namely, individual parameters, properties, and method calculations may change over time. It is not only about internal computation and performance optimization but also about ways of processing and calculation. Thus, the function is not deterministic across the time axis. Existing solutions are primarily based on versioning considering the function name, which, however, takes significant risk related to the reference correctness as it depends on the function name. Precisely, there is no checking of the validity referencing specific function versions. The proposed solution aims to prepare an automated, robust transformation and mapping solution that emphasizes individual versions of the functions and calls relevant functions based on the referenced time frames. Thanks to the provided reliable transformation layer, multiple function versions can exist, respecting the valid version of the function that existed at the time of validity of the relevant tuple. The function is always called by the original name, and a particular version is automatically processed based on the input data and referred time frame. In the performance evaluation process, we focus on three stages – pre-processing phase (parsing, checking, and loading), version identification, and loading. Multiple solutions were created and discussed. The best results were provided by model 4 based on the transformation module in the temporal layer, for which the total cost of processing within the Select command, compared to the reference solution, was reduced by 68.40%. The most significant saving was detected in parsing and checking since the function versions are directly available for the reference calls (72.96%). Version identification is done by the optimized searching B+tree index tree, reducing the costs by up to 69.42%. The proposed solution is based on managing multi-versioning by propagating new clauses, commands, and background processes, managing the whole system autonomously. Furthermore, performance optimizations can later enhance individual versions, like more sophisticated data structures. Thus, each version can have multiple implementations. The additional demands on data storage and structure management are less than 1% globally.

INDEX TERMS Temporal databases, function versioning, dynamic function version mapping, relational database, performance.

I. INTRODUCTION

Currently, databases are an integral part of any information system. Even simple monothematic applications use

The associate editor coordinating the review of this manuscript and approving it for publication was Vlad Diaconita.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

databases to store data, parameters, or configurations. Either way, database systems must provide efficient storage for disk space consumption and access and processing of complex queries. Data in the relational schema is stored in the form of blocks [1], which are accessed either sequentially or by using indexes [2], [3].

Data was conventionally oriented in the initial phases of the relational paradigm of the database system definition and development. It means that only current valid states were present in the database. Any Update operation expressed the replacement of the original state with the new version [4]. Ideally, a new state is placed on the original block; however, data migrations can be present due to a lack of space in the block, referring to the new block from the original repository using the data pointer [5], [9]. It demands additional processing since the index always points to the original data block. However, the row tuple is not present there. Data migrations do not influence sequential data block scanning. During the development of database technologies and the SQL paradigm, the concept of temporal databases was released. Since then, it has been possible to store the individual states of objects over time and thus monitor their changes. Thus, the time frames encapsulate each data row, mostly expressing the validity range. Consequently, one object state can be expressed by multiple rows. However, only one state can be valid at a time.

With the evolution of the objects and individual state representations, it is worth dealing with the function references and their time evolution. Namely, one function body can also evolve over time, not only in terms of optimizing code to run faster but also in terms of producing results. The calculations and data treatments can evolve, and the same data inputs can produce several outputs based on the temporal function versions. Thus, it is necessary to build a robust apparatus for treating function versions and apply relevant versions for the call in a complex environment enhanced by various source joining [4], [8].

This paper analyzes existing approaches to function versioning on the database layer by pointing to the limitations of the current approaches. The paper´s goal is to create an automated versioning system. The principles are analogous to what cloud file repositories allow but at the level of a parsed version of functions with the possibility of calling them immediately.

The motivation is to maintain changes defined inside the function, either in form of code optimization, but mostly to preserve temporal changes of the implementation. In case of changing function content, list of parameters can be changed, but primarily, the mapping between input and output values is changed. Although it is primarily focused on the transportation systems, in which the evolving characteristics, parameters and environment reflection must be considered, implemented techniques and proposed solutions can be generally applied in any field of informatics and intelligent information systems, by treating patients and helath monitoring or in environmental data analytics.

The paper's main contribution is a set of models for function version management inside the Oracle database. Each solution is critically described by pointing to its properties, characteristics, and limitations. Proposed solution 4 seems to be the most promising in terms of performance. The core of this model is the transformation mapping module specified by the temporal layer.

The paper is structured as follows: Section II defines the problem and specifies the paper's main research strategies and contribution. Section III provides a brief summary of the temporal database approaches by emphasizing architectures and granularity precision management. This section is essential since it serves as the initial background for the proposed solutions.

Section IV deals with function references and their applicability in database systems. It points to parsing and caching, indexing, and pre-storing function results in virtual columns or analytical-oriented database architectures. Section V points to the temporal functions and referencing of the individual function versions. Proposed solutions are introduced in section VI by emphasizing the mapping and version transformation process, spreading the solution's applicability to the vast world of parallelism and data distribution. A performance evaluation study is presented in section VII. Finally, findings, conclusions, and future research perspectives are stated in section VIII.

II. PROBLEM DEFINITION, PAPER CONTRIBUTION

Functions are an inseparable part of the programming. SQL is a non-procedural language, but the environment always supports procedural extension, which refers to the execution code step-by-step. To serve that, PL/SQL language is supported by the Oracle database used in this paper. However, any database system generally provides procedural extension support, which can be applied. Methods referring to the procedures or functions are conventional [6]. This means that one version of the code is present in the system in a parsed form, which is then loaded in case of referencing. Thus, if a new version is to be compiled, the original method definition is replaced by the new one. Precisely, there is no version history. From the database system point of view, there is no evidence about the historical versions; source code of the previous versions cannot be located.

Reflecting on the temporal environment, it is worth mentioning function references. The definition, implementation, and function results can evolve over time based on the laws and regulations, changed environmental characteristics, etc. Imagine the evolving characteristics and parameters influencing the output. Precisely, for the production monitoring, a few parameters were evaluated, however now, many precise sensors and data are considered. Or consider the environmental data standards and impacts, which are getting stricter every year. Thus, it is clear that the function content is evolving, not only from the efficiency of the implementation, but it can strongly relate to the changing number of parameters to be considered, as well as the mapping input into output

values. Another example can be associated with calculating salary and tax, which may change in individual periods. The function remains the same (parameters, name, header, etc.), but its internal rules have changed.

To make the system relevant and applicable in a temporal environment, it is necessary to consider time-evolving characteristics and map the temporal approach into function references [7]. That´s the reason for incorporating the temporal paradigm and summary of the temporal approaches at the data architecture level, stated in section III.

To make the system robust, it is necessary to create a compelx solution that applies the appropriate version dynamically and automatically based on the selected temporal framework.

Currently, there is no general approach and defined rules for processing function version code, e.g., compared to the GIT approaches used in common programming languages. Although many partial solutions exist, they are mostly related to the specific application domain or do not provide a robust and secure layer. In [7], temporal data mining using functions is discussed. Techniques for handling periodic facts is defined in [6].

Individual versions differ primarily in the header of the function and their names. Consequently, each user must be aware of individual versions and call suitable versions based on their unique name. However, such a premise is not suitable, while the method version selection depends on the user's knowledge. Furthermore, there is no control mechanism to check the selection of suitable candidates. Therefore, often, only one code version is maintained, covering individual temporal frames inside. However, there are three negatives to mention:

- Source code becomes more and more complex, and even parsed versions of it are so huge to be maintained by the database – loading, and handling becomes more and more time-consuming over time since new code is implemented in one bulk document.
- It is always necessary to preserve the current source code for the function, otherwise changes done in the past could be lost, consequencing in the reliability breaches.
- During the process of compiling new implementation of the function, original one is unavailable, limiting the performance and accessibility inside the calling information system.

The paper aims to answer the following challenges:

- How to optimize and split code into individual versions?
- How to manage versions and reflect the validity time frame?
- How to limit function unavailability?
- How to secure versions by automatic mapping?
- How to ensure parallelism, synchronization and scalability of the whole infrastructure?
- How to build robust layer considering automatic version mapping?

Based on the previous discussion, it is clear, that robust solution dealing with the function version must be developed. Section IV deals with the existing solutions and workarounds to manage temporal version references. Section V points to the proposed solutions, which are not inly introduced, but also critically discussed, supervised by the performance evaluation study present in section VI. Computational study focuses on three aspects:

- process of function version parsing, checking and compiling,
- identifying version,
- version control loading.

Four proposed solutions are presented, discussed and evaluated. The used data set relates to the environmental data monitoring flights. As described in section VI, the best solution is associated with the temporal model, for which each version is directly associated with the validity time frame. The used temporal background layer is bi-temporal. In the used environment, total processing costs were reduced by almost 70%. Specifically, for the parsing process, it reflected a cost reduction of nearly 73%, version identification reflected a 69% reduction, and for version loading and application, costs were reduced by nearly 54%. However, it's not just about numbers, the primary goal is to create a dynamic mapping infrastructure where the system automatically selects the correct version of the feature. Thus, the user only defines the call of the temporal function, the actual selection of the temporal version is done automatically by the database system itself.

The above performance reflected the data retrieval process. However, the additional costs of maintaining the entire ecosystem should also be emphasized. Based on the evalution, added cost increase is expressed at the level of 0.89%.

III. TEMPORAL DATABASES

Temporal databases form an extended evolutionary step of the relational paradigm. Instead of storing only current valid values, in which the object is identified by the unique identifier - primary key, temporal databases point to the object itself and multiple versions valid during the defined time range [9]. Thus, the original object identifier is no longer sufficient, it is extended forming the composite definition – object identifier, validity time range, and data themselves. Thus, if there is any change in the object's state value, a new state must be composed as an inseparable element [10]. To be precise, all data attributes must be filled in. If the object update does not change all data attributes, original values must be copied to the new state definition [11], [12]. Although the state identification is straightforward and easy, there can be many duplicates and real change reflection is hard to locate [13], [14].

Another approach states the table column as a processed precision [17]. The used model is called an attribute-oriented system (Fig. 2). It is based on associating temporal dimensions to each attribute separately. The advantage of the solution lies in the ability to process each attribute separately by the update operation. Thus, no duplicate tuples are present, and the database layer is optimized. On the other hand,

additional data demands are present, caused by the temporal time frames referred to by each attribute. Furthermore, getting any state requires combining individual attribute values.

The synchronization groups cover the intersolution between object-level and attribute-oriented granularity management. The change operations define synchronization groups. They can be detected automatically, or manual selection can also be done. Each synchronization group is temporarily defined and consists of individual attributes or groups that can be set as the subelement. Fig. 1 shows the data model for the group definition.

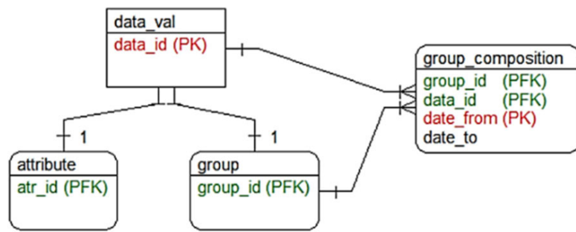


FIGURE 1. Group-level temporal architecture – data perspective [17].

Data_val expresses a synchronization group, which is then handled as a single unit. Thanks to that, the temporal layer does not reference individual attributes but synchronization groups in the event of change.

Fig. 2 shows the architecture of the solution. Compared to the attribute-oriented approach, an additional group detection module, composition, and drop is created. Instead of reflecting attributes, data_val values are referenced.

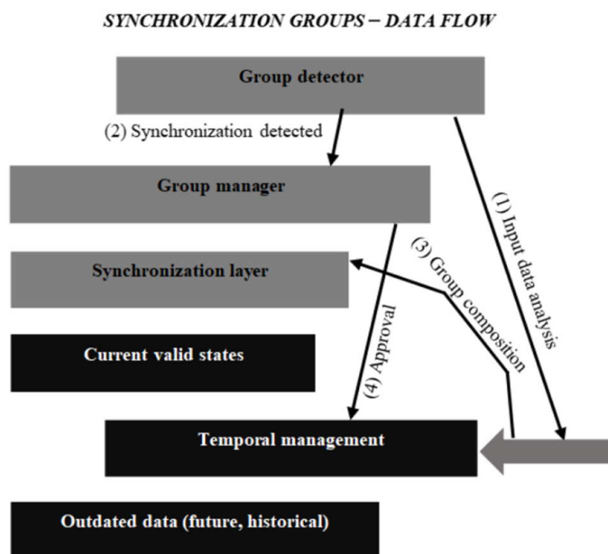


FIGURE 2. Group-level temporal architecture – architecture [17].

Temporal databases do not need to define the time elements precisely strictly. In [18] and [19], fuzzy temporal databases focus on the relational model transformation and retrieval process. Spatial enhancements of the temporal paradigm are listed and discussed in [20], [21], and [22]. Temporal data clustering for fuzzy association is covered by [7], followed by the function pattern definition discussed in [24] and [25]. Although we are not directly considering a distributed environment in this paper, it is worth mentioning temporal models for sharing function content in that environment since it will be an integral next step for research and development. The core principles and implementation details can be found in [28], [29], and [33]. RDF/OWL storage and management in relational database management systems considering data distribution is discussed in [34]. All the mentioned solutions point to the need for proper data sharing in a distributed environment. These concepts can be applied not only to the data itself, but were also prepared for functions in a conventional (non-temporal) environment. Our future goal is to propose a methodology for the distribution of temporal function versions. We deliberately do not deal with it directly in this paper, because a prerequisite is the correct and robust design of the data layer at the level of one node, as presented in the proposed solutions.

IV. FUNCTION MANAGEMENT AND REFERENCES

SQL is a non-procedural language that specifies the data to work with, but there is no mention of how to access the data, locate them, where they are physically stored, etc. All these tasks are part of the database system decisions. In contrast, each relational database system covers the procedural extension, too, in which a person strictly specifies individual operations and flow step-by-step. The name of the associated procedural module can vary based on the database system type. For the Oracle Database, which will be used in the computational study, procedural language is the so-called Procedural Language for SQL (PL/SQL). It primarily focuses on the methods – procedures, and functions- but also packages them as a group of methods, triggers, anonymous blocks, etc. Regardless of the type, it is a sequence of individual commands, navigations, conditions, and routines. Each method consists of the header, parameters (optional), local variables (optional), and body between *BEGIN* and *END* commands. Usually, the code is written to the files, stored in the file system outside the database, and a particular function is then compiled [23], [27], [28].

Compilation itself is a staged process. First, it must be checked whether the syntax and individual routines are valid. If so, the source code is transformed into the parsed compressed version, readable by the database system and optimizer. This version is stored in the system structures, and individual parameters and properties are extracted to the system tables. A pre-stored system version is used for the recompilation, so the function definition source stored in the file system is no longer necessary. Fig. 3 shows the flow.

When using a call, an already parsed and prepared version is used. Oracle Database can even change the order of the commands inside the method and rearrange it (if enabled by the system parameter PLSQL_OPTIMIZE_LEVEL), providing an additional optimization level. Inside the function

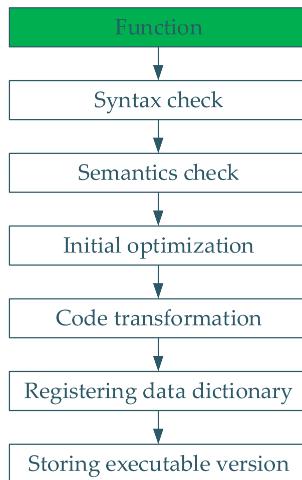


FIGURE 3. PL/SQL code compilation - flow.

definition, references to the other objects (tables, views, methods) are extracted. Therefore, if any referred object is used, a particular method is automatically flagged as invalid, forcing the system to recompile it at the latest at its next call.

To execute a function, it is necessary to load the parsed definition from the database to the instance memory (structure Library_cache) if it is not already there. Then, the input data values are obtained, and the function can be executed, producing the defined output. The problem is that the function can be robust and require huge system resources, and it can last several seconds, even minutes, or hours for complex analytics [29], [33]. If the same function is called several times with the same parameters, the same code must be executed multiple times, right? To limit that, a function can be marked to navigate the system to store the pair–input values and resulting output in the Result Cache structure of the database instance memory. Access and mapping is almost immediate since it is in the memory (currently delimited mainly by the fast NVMe memories). The method is defined by the unique identifier originating from the system tables, or a secure hash value can be used.

Result Cache memory structure is sophisticated and reliable, so if the function is structurally changed, pre-stored values are automatically marked as irrelevant and deleted from the Result Cache memory structure. The data flow from the calling environment point of view is depicted in Fig. 4.

Function references and calls are relevant parts of the data management layer and can generally be treated similarly to the attribute values. So, they can be indexed by forming a function-based index key (based on the assumption that the function is deterministic); function results can be physically stored in the analytical structures, providing a de-normalization layer and improving data access and evaluation perspectives and performance. Furthermore, materialized views can hold data physically based on complex queries and function calls. If the materialized view log is present, a fast refresh can be done automatically based on the source data changes, so it is ensured that the materialized view always provides up-to-date values and reflects any data change.

The following section points to the temporal extensions and function version management.

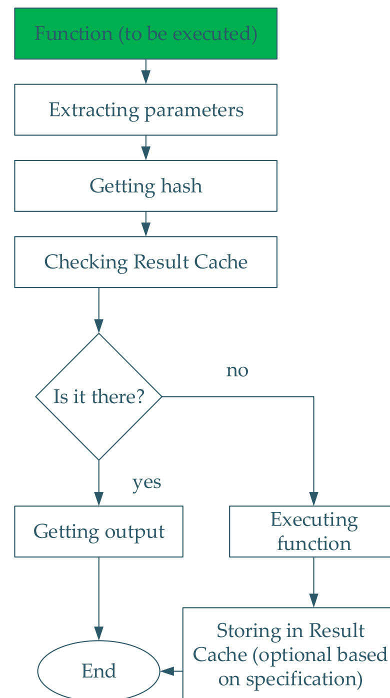


FIGURE 4. Executing function.

V. TEMPORAL FUNCTIONS

The idea and concept of temporal functions and version reflection were created soon after the first release of the object-oriented temporal model. It was clear that individual function versions would need to be treated sooner or later. To serve that, the function version system needs to be placed in the database system world. Unfortunately, the already provided versions cannot be considered robust, and particular partial solutions are rather logical concepts that are physically implemented and built.

The core concepts were based on source file versioning and dynamic loading. Thus, for each function, several code source files were present; each file consisted of the header-defining version (commonly associated with the validity time frame). However, such a concept had many drawbacks. First, it was necessary to ensure version consistency and that there was only one valid version at the time. On the other hand, it was necessary to identify time frames when the version was not defined, invalid, or the code could not be successfully compiled. This additionally brought the fundamental requirement of version change management. Namely, if you compile a method, it does not automatically mean that a new version needs to be released. Each new version should be associated with the implementation change regarding the data handling

and outputs. Thus, if only code optimization is done inside, it does not mean the new version will be created.

The preliminary concept and version management are shown in Fig. 5. There are source files present in the file system. If the function is to be called, a particular version is loaded, including all checking (syntax, semantics, etc.), parsing, notifying system tables, recording the change, and the execution process itself. Thus, it is clear that significant additional steps are necessary, forming enormous resource demands and time consumption. As evident, it limits the parallelism since only one version can be in the system at a time. All these switches are necessary to form the main limitation and applicability of the solution.

The initial solution was slow and resource-intensive, and the entire management and reliability treatment was complicated. If the code modifications were implemented only from a performance point of view without creating new versions, the whole solution would be challenging to manage. It was simply too complicated, bringing significant additional costs [34].

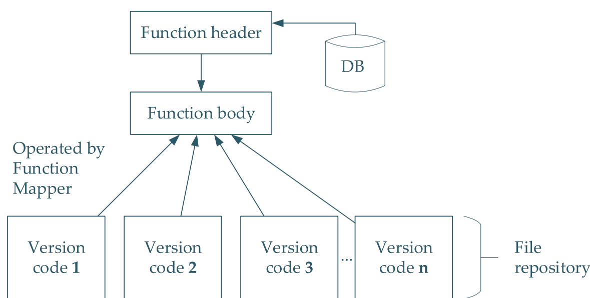


FIGURE 5. Version codes.

In addition to the already mentioned shortcomings, it should be noted that the individual source code had to be kept in the form of files, which, however, were stored outside the database. The database system itself did not manage them or provide any security layer for them. Instead, it was the task of the developers to ensure the availability and security of these files, as well as the monitoring of changes, versions, and accessibility in case of moving to another repository, disc, etc. [27], [35], [36].

A. BI-TEMPORAL SOLUTION HANDLING FUNCTION VERSIONS

All the mentioned applied principles led to creating a bi-temporal approach to managing function versions. The first level took individual versions, which influence output values (the structural body of the function was changed), e.g., the principles of the tax calculation were changed. The second level held changes in the implementation, which, however, did not impact the mapping between input and output. That is, it is only a matter of changing the effectiveness of the implementation, for example, by changing the data structure, optimization of query processing, and so on. Such a change mustn't affect the output. So, it does not refer to a new version, just a change in processing efficiency. The architecture is shown in Fig. 6. Layer-holding versions refer to the timeline or transaction identifiers that can be used (commonly expressed by the System Change Number (SCN) value). The implementation layer does not need to refer to the validity; transaction references, correction identifiers, or simple sequential values can be used. In principle, if the function has several implementations and can be used, the only difference is in the performance and system resource consumption perspective.

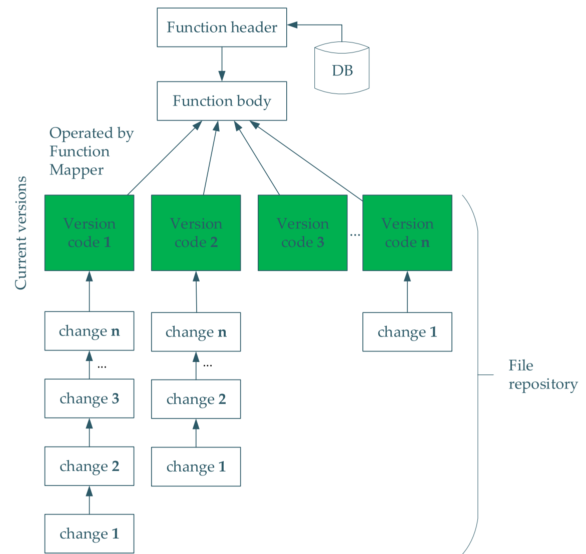


FIGURE 6. Treating versions in a linked list.

B. SOURCE DATA FILE MANAGEMENT

Based on the previous analysis, it is evident that storing source code outside of the database is not reliable since there is no supervision of the source files. Furthermore, the database system cannot directly locate the files for security reasons. To make a workaround, a database directory object must be created. Privileges to the database user must be set on the database and the operating system file (to make the files accessible). However, it is still impossible to fully control the changes, identify new versions, and locate them because the actions are out of database management. In [8], function code services were moved into the database by introducing a system table storing code files in the large objects (LOB) format. To serve that, character or binary LOBs are used [8], delimited by three elements:

- Function identifier (Function_id) obtained by the object_id column of the dba_procedures data dictionary. For the package, individual functions need to be referenced. Therefore, the identifier is extended by the subprogram_id attribute (for standalone functions, subprogram_id attribute values always value 1). Please note that the name of the function stated in the following query is only for representative purposes, and the name

itself is not considered an identifier for the version reference:

```
select object_name, object_id, subprogram_id
 from dba_procedures;
```

- Temporal attributes expressing validity, such as temporal references, are commonly managed by the bi-temporal principles stated above. Validity can be expressed by two attributes: delimiting the start and end points in the defined representation [17] and allowing the provision of undefined versions at some time. If only the validity's begin point (BD) is defined, then undefined representations must be stated in a specific way, commonly with no reference to the function code content.
- Content—the source code for the function, extracted by the 1:1 mapping—one source file consists of one version of the function.

This structure makes it easy to maintain efficiency and references to individual versions. Moreover, all existing temporal approaches can be used, whereas the table is temporal (reflecting object-level temporal architecture). However, what about the limitations and real applicability? First, creating binary objects from the source code is impossible, so the additional mapping layer would be inevitable. Moreover, the code remains original, so the syntactical, semantical checks must always be present. Mapping between the database and source files must be present. Finally, there is no way to use parallelism because, before the call, the particular version must be loaded, parsed, and made valid. Fig. 7 shows the basic architecture and data flow.

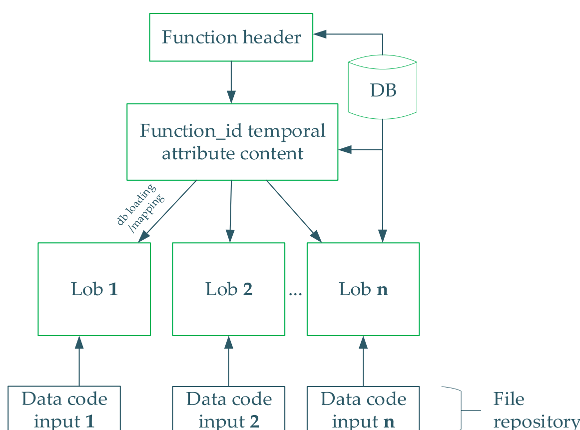


FIGURE 7. LOB holding function versions.

In conclusion, storing source function data directly in the database brings only a slight security level, so the transaction manager can identify versions and load them smoothly without contacting the external repository. However, the problem of parallelism is not solved. Releasing is too demanding and resource-consuming - all loading steps must be done, and they are mostly represented by various checks and optional transformations.

C. STORING PARSED DATA IN THE DATABASE

The stated source data file management extension is defined by storing pre-parsed functions. Compared to the original solution, checking, parsing, and other keeping activities can be limited since they have already been done once at the beginning. Naturally, if there is a change in the referenced objects, such function versions are marked as invalid, and the system is asked to compile them during the next call. Consequently, apart from the parsed versions, which can be directly loaded and then run, the original source code (in the format of LOBs) must be present to compile the method after any reference change. Still, parallelism is restricted for each temporal function call caused by the necessity of loading.

D. MULTIPLE FUNCTION MAPPING NAMES

The last solution presented in this section originates from the previously mentioned concepts. However, the main emphasis is to remove the previous limitation in the necessity of sequential processing of function calls. Considering that many variants (versions) of a given function can be called within one Select statement, constant needs for loading and mapping and the associated costs are apparent. The individual records in the data processing are not sorted in any way by default. So, it would result in swapping between versions and would cause the necessity of multiple calls of the same version within a single query. One of the solutions defined in [17] is to create partitions based on function feature versions. If there are several temporal functions in one query, subpartitions (subsections) are created, unlimited to the number of levels. Thanks to that, the function is loaded into the system for each leaf node of the partition only once. In addition, the system tables store the implementation cost for each version of the function, making it possible to appropriately choose the order of the functions in the partition hierarchy.

This solution is based on versioning functions through their names. All versions are installed (pre-loaded) directly in the system, but the user usually only calls the root function (without version specification). The original code, therefore, needs to be dynamically parsed and replaced. However, it has significant disadvantages, mainly from the point of view of indexing and searching. Specifically, from the database layer, it calls a different function (delimited by a different name expressing the version), so the index would not be used! In practice, this means that, on the one hand, the possibility of parallelism has been significantly increased; on the other hand, there is a performance problem related to indexing and searching.

The entire system management is based on correctly mapping functions and their versions through created background processes that dynamically ensure the correct assignment of versions. Function_version_assigner_n worker processes are allocated in the system. There is at least one such process, and the total number of these processes is managed by the Memory Manager system resource so that individual function

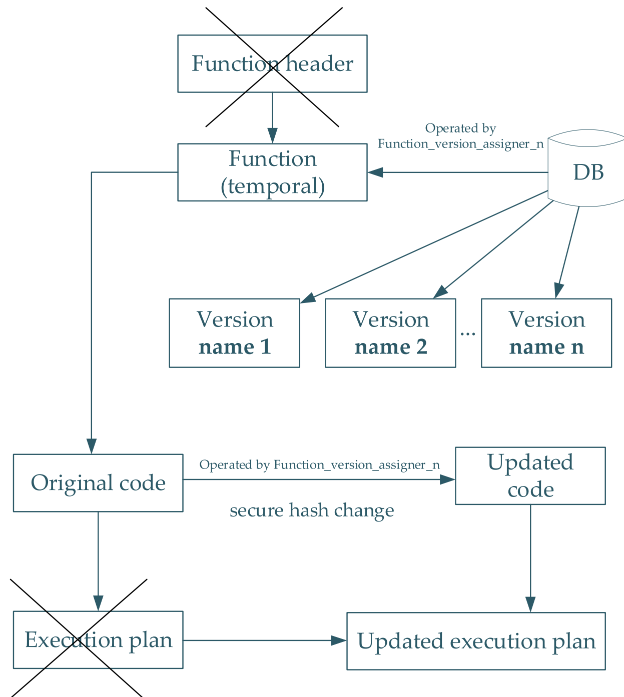


FIGURE 8. Mapping names.

mappings are processed immediately without unnecessary waiting [37]. Architecture is shown in Fig. 8.

In conclusion, differentiating function versions enables temporal function management parallelism. However, it limits the performance since the stated queries reference different functions enhanced by the version definition. After all, whereas the original query would be structurally changed, the original execution plan identifier provided by the hash function would also be changed. Although it could be solved by hashing the original statement instead of transforming, individual temporal function versions would not reflect relevant functions enhanced by the temporal version control.

VI. PROPOSED SOLUTIONS

State of the art and analysis [38] have shown the direction and focus on the consecutive development and research perspectives. In our proposed solution, the following aspects are primarily handled:

- Enabling massive parallelism by limiting parsing, loading, and transformation necessity across the versions.
- Dynamic version mapping, in which one temporal function header remains the same for all versions.
- Differentiating testing and officially released versions, identifying and registering versions to limit test cases (if the recompilation does not change the implementation of the function nor create a new version, it is not necessary to register and process the given function separately. It can be merged within the scope of validity of the previous version.
- Handling version implementation updates to improve performance – the goal is to ensure that the most efficient implementation of a given version is always executed.
- Registering a new function version, which will become valid at the defined time point.

The architecture of the proposed solution is depicted in Fig. 11. The Temporal_version_loader background process checks individual versions. This process ensures the loading of the version and proper mapping of the implementation performance enhancements of the individual versions, emphasizing the validity of the individual versions. Additionally, there is a Temporal_registrator process that is responsible for recording a new version in the temporal layer managing functions. Furthermore, it checks the consistency of the temporal frames to limit any version overlapping over time.

The temporal function manager provides those functionalities:

- Register temporal function.
- Register new version:
  - valid immediately,
  - valid in the future.
- Register version implementation change.
- Recompile registering new version/version implementation change – transforming existing functionality into a new temporal element.
- Disable ALL – temporal function no longer accepts any new versions nor implementation changes for the particular version.
- Disable versioning—The temporal function no longer accepts new versions, but the implementation of the existing version can be changed to improve performance (but it does not influence the relationship mapping between input parameters and provided data outputs).
- Merge versions—This functionality is used when compiling code several times, which creates new versions. However, if the code is still the identical, consecutive versions can be merged.

The data layer for the temporal function registration is in Fig. 9. The temporal database consists of the function identifier. Whereas the package, object_id and subprogram_id can frame it are stored. Then, there is a temporal reference, defined by the beginning point of the validity. Thus, each new state automatically limits the definition of the previous version. On the other hand, there can be situations where the version should be applicable only during the defined time frame, but the next version is unavailable. To serve the correct limit of validity (ED), a specific function version is registered, expressing that the code is unavailable but that the existing version can no longer be used. Then, there is a sequence for performance implementation change control. Finally, there is an object reference to the parsed version of the source code. Thus, syntactical, semantical, and privilege checks are done only once during the initial parsing and loading. The object

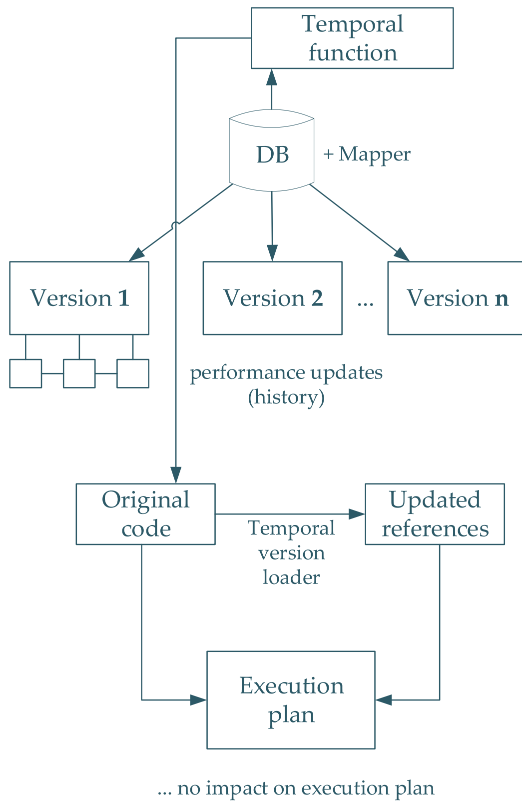


FIGURE 9. **Proposed architecture.**

owner and authid method are stated (definer or current_user). The data model is shown in Fig. 10.

Object_id and subprogram_id form the identifier of the method. BD reflects the first level of temporality. Perf_update_seq refers to the second level of the temporality. The executable function version is accessible via object_reference.

Temporal_function_management			
object_id	Integer	NN	(PK)
subprogram_id	Integer	NN	(PK)
BD	Date	NN	(PK)
perf_update_seq	Integer	NN	(PK)
object_reference	Char(20)	NN	

FIGURE 10. **Temporal function management data layer.**

It is a bi-temporal model dealing with the validity of the versions (the first temporal layer) and reflection of the performance implementation changes (the second temporal layer). It can, however, happen that the existing version is later identified as incorrect and must be replaced. The system must record that an improper version was released in the past since it could be used for analytics, evaluation, and processing. Thus, the proposed solution can be extended to the multi-temporal system by identifying three temporal dimensions:

- version validity,
- performance implementation changes,
- version corrections (modeled by the transaction beginning point, based on the concept that the current version is considered to be correct in an unlimited manner).

A. DATABASE STRUCTURE IMPACTS

The main advantage of the proposed solution is that it uses logical version identifiers, which are part of the object reference address. Thanks to that, the name of the method remains the same. The user-defined code is not changed (e.g., by using translation operation applying the correct function version, as used in the existing solutions). Thus, existing execution plans registered to the defined statements remain the same and must not be evaluated. Furthermore, the code is not version-dependent because the correct version is applied dynamically based on the data. Therefore, if a new function version is released, the source code is not impacted. Another aspect relates to structural optimization to improve performance. In principle, the newest method is considered the most effective. Thus, the version with the maximal sequence_id is selected to be executed.

As evident from the above specification, the database layer and optimization have direct access to all the versions of the functions, so there is no need to parse, locate, load, all the required data are already present in the database system tables and directly accessible using the stored object version pointer. Thus, the only activity is to load the content of the function version from the database into the instance memory if it is not there yet. The various properties can be obtained from the dba_procedures or generally dba_objects based on the type of condition:

```
select * from dba_procedures;
select  *
  from all_objects
   where object_type
     in ('PROCEDURE','FUNCTION','PACKAGE');
```

The dba_arguments data dictionary view can obtain parameter specifications. Object_name refers to the name of the root method, and package_name deals with the identifier name of the method's optional encapsulation using the package.

```
select a.*, object_type
 from dba_arguments a
    join dba_objects o
   on(a.object_id=o.object_id)
  where object_type]
   in ('PROCEDURE', 'FUNCTION', 'PACKAGE');
```

Object_id is a unique method identifier. Argument_name refers to the parameters delimited by the data_type attribute and position. The positive value of the position attribute defines the order of the parameter in the input definition. Value 0 of the position attribute refers to the type to be

returned. In that case, argument_name holds a NULL value. A complete list of individual columns, data types, and descriptions can be found in [35].

B. REGISTRATION AND IMPLEMENTATION COMMANDS

In this subsection, implementation commands will be stated. Three levels of applicability can be used – system level, session level, and object level. There are two parameters to be set:

- temporal_registration, which defaults to FALSE. It expresses the behavior of the new function definition. If set to FALSE, such a new function does not create individual versions and is treated as conventional (if a new version is to be compiled, it replaces the original solution. Thus, no evolution is present for the function).
- temporal_versioning is applicable for the function, which is temporal. By default, it holds a TRUE value, expressing that each new compilation of the method automatically releases a new version. A new version must be stated and registered explicitly if set to FALSE.

The server precision level is always set, or default values are applied, respectively. If not set explicitly, session-level values inherit the values from the server. A similar approach is used for the object level. An object is characterized by the method itself.

The object level setting can be defined during the compilation by the **PRAGMA** keyword or **alter object ...** can be used and enhanced by the setting object parameters. For simplicity, the following stated code snippets refer to the function definition; procedure management is similar. However, there is no return value.

C. FUNCTION REGISTRATION

Using PRAGMA_TEMPORAL_REG ensures that the newly created function will start creating individual versions and is temporally registered. If the function is already present in the system and has at least one version, the specified PRAGMA is ignored since it has already been set to register temporal versions. Vice versa, by using this PRAGMA definition, it is possible to transform conventional (non-versioned) functions into temporal ones. Physically, it invokes the introduced background process by checking the temporal validity frame and register function.

```
create or replace function function_name
 return data_type
    is PRAGMA_TEMPORAL_REG;
         [local variables definition]
- body:
begin ... end;
/
```

D. VERSION REGISTRATION

During development, it can be useful to create various versions, recompile solutions, etc. In that case, it is not necessary to create new versions. Similarly, some versions can be automatically replaced, with no reflection of temporality, treated as an optimization of the existing code with no reference to the change of the input and output value mapping. The new version can be registered using the PRAGMA_VERSION_REG keyword.

```
create or replace function function_name
  return data_type
     is PRAGMA_VERSION_REG;
                    [local variables definition]
- body:
begin ...end;
/
```

E. VERSION MERGING

It can, however, happen that one code is recompiled multiple times, creating multiple versions. By using the following command, individual versions can be merged. It is assumed that all versions are equal. Physically, the first version is retained, and the others are removed. Individual versions can be identified by their names, or the validity time frame can be stated. Reflecting the performance updates of one version, the newest one is always used.

```
alter function function_name MERGE VERSIONS
 [ {(list_of_version_identifiers)]
    | (BD, ED)} ];
```

Please note that only versions which are entirely covered by the validity time frame are covered:

```
select versions
from the temporal_function_management
 where process_temporal_frame
                    between BD and ED;
```

The process_temporal_frame function stated above produces a validity range for the particular function version. It takes the local compilation timestamp. The end of the applicability is calculated based on identifying the next version, which limits the validity of the predecessor.

F. MANAGING TEST VERSION

Test versions are a specific category of temporal function management. If the code is flagged as TEST, then it is not temporally treated; no new version or performance update is registered. It is primarily used in a development environment to check the correctness of the new code. Such a function can be compiled, but its original name is enclosed in the test version:

```
create or replace function function_name
 return data_type
    is PRAGMA_TEST;
         [local variables definition]
-- body:
begin ...end;
/
```

If the test version is not explicitly marked, then it is assumed that performance optimization of the existing version is to be set.

G. RECOMPILATION

The already created and compiled code can be enhanced by the temporal registration using temporal or version options. TEMPORAL_REG asks the system to register a function to be temporal and create a version, while VERSION_REG creates a new version for the function, which is already marked as temporal.

```
alter function function_name compile
   {TEMPORAL_REG | VERSION_REG};
```

H. PLANNING

The above principles assume that the applicability of the new registered version becomes immediate after the compilation. An optional parameter of the PRAGMA clause offers to set the precise beginning point of the validity, either by the Interval or by timestamp specification. The value must refer to the future. If not, an exception is raised, and a new version is not compiled and registered:

```
alter function function_name compile
 {TEMPORAL_REG | VERSION_REG}
   VALIDITY {TO_DATE(input_val,
                 'DD.MM.YYYY HH24:MI:SS)
              |
                 INTERVAL 'value'
             {DAY TO SECOND | YEAR TO MONTH}
                        };
```

I. REGISTER PERFORMANCE UPDATE

This clause allows you to optimize the performance of the existing version either by applying new features, creating new sophisticated methods, optimizing data flows, etc. By default, the current version is optimized, but the definition can be enhanced by stating the version identifier:

```
create or replace function function_name
 return data_type
    is PRAGMA_PERF_REG [ version_identifier];
          [local variables definition]
-- body:
begin ...end;
/
```

If the given version identifier did not exist, processing would end with an error.

Concluding the temporal function management, available states of the temporal function are listed in Tab. 1:

VII. PERFORMANCE EVALUATION STUDY

The performance evaluation study was done using the flight data monitoring data set. It takes the flights made over European space between 2018 and 2022. The ECTRL_ID attribute can uniquely identify each flight. Associated measurements and flight parameters are then delimited by the occurrence timestamp. In addition, the values are ordered in time through the SEQ_ID attribute. The data set consists of 100 flight parameters and planned and actual routes. It covers the flight monitoring from the taxi through the take-off, flight itself, up to landing, taxi, and parking. Besides, flight information regions (FIRs) can be identified and associated based on the positional data. In aviation, a flight information region is a specified region of airspace in which a flight information service and an alerting service (ALRS) are provided. Each FIR is part of one air navigation region: Africa–Indian Ocean Region (AFI); Asia Region (ASIA); Caribbean Region (CAR); European Region (EUR); Middle East Region (MID); North American Region (NAM); North Atlantic Region (NAT); Pacific Region (PAC); and South American Region (SAM). FIRs vary in size. Smaller countries may have one FIR in the airspace above them, and larger countries may have several. Airspace over the ocean is typically divided into two or more FIRs and delegated to controlling authorities within countries that border it. A significant aspect of the FIR is based on its evolution over time. Namely, the borders and positions are not strictly set and are periodically evaluated. Thus, their definitions also evolve. In this environment, individual flights are monitored to calculate the flight efficiency, fuel consumption, costs, and impact on the environment. Currently, the problem is even sharper, it is strictly inevitable to reduce the impacts on the environment. This data set was also selected based on the supporting projects of this research – analyzing environmental data using the EverGreen project, as well as researching and optimizing intelligent transport systems (VEGA 1/0192/24). The data set consists of 500,000 flights with various lengths and frequencies of obtaining the data. In total, the data set takes one planned route and one real route for each flight, covered by 100 parameters (80 are real, 10 are integer, five are textual 4 are express time references, and 1 is JSON). On average, each flight contained 1000 data rows (respectively, the data set itself was modified in such a way by reducing the data sampling rate).

An example of the data is shown in Fig. 11.

The parameters of the server on which the performance evaluation study was done are stated in the following paragraph:

TABLE 1. Results – existing approaches.

Status	Monitored temporally	New version released	A new performance update of the existing version was released	Not registered
Conventional	×	×	×	×
Temporal + version	✓	✓	×	×
Temporal + performance update	✓	×	✓	×
Test	✓	×	×	✓

```
"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 11. Data.

- Processing unit: AMD Ryzen 5 PRO 5650U, 2.30 GHz, Radeon Graphics
- Memory: Kingston, DDR4 type, 2 × 32GB, 3200MHz, CL20
- Storage: 2TB, NVMe disc type, PCIe Gen3 × 4, 3500MB/s for read/write operations
- Operating system: Windows Server 2022, x64
- Database system: Oracle Database 23c, release bundle Oracle 23c Free, Developer Release Version 23.2.0.0.0.

The performance evaluation study focuses on the following aspects, processed by the temporal functions:

- **F1:**This converts the local timestamp into the UTC format by respecting summer and winter time. The calculation is changed for each season. To be strictly precise, it also applies leap seconds.
- **F2:**Getting FIR assignment based on the positional data.
- **F3:**Evaluating the impact on air pollution by respecting the time-varying parameter limits.
- **F4:** Evaluating flight efficiency by comparing the optimal and real route by respecting current weather conditions and other restrictions.
- **F5:**Complex flight monitoring produces JSON, characterizing the route (planned and real), flight efficiency across individual FIRs, and flight parameters across all regions.

The general rule for the provided computational study is to experimentally analyze the performance impacts of the staged compiling process, delimited by the loading process, to enable proper function version execution.

For the evaluation, five solutions are referenced:

- **REF** refers to the existing solution in which the temporal versioning has no specific support; individual versions are located in the files. To get the proper function, a particular version needs to be identified based on the code of the function header, followed by the parsing, checking, loading, and executing.
- **SOL1**brings an additional security layer by storing the code of the versions in the database. However, the function code still needs to be parsed, checked, and pre-processed before execution.
- **SOL2** removes the necessity for parsing and checking because the version is stored in a parsed version done once during the initial processing.
- **SOL3**offers accessibility to all versions, which differ in their names. All the versions are directly available for execution; however, it is necessary to structuralize provided code definitions by replacing the original function name with the applicable function version. Consequently, whereas the code definition is changed, execution plans that have already been created are not associated and cannot be later referenced. Thus, each function version produces a different Hash Plan ID value.
- **SOL4** describes the proposed solution by introducing a transformation mapping module specified by the temporal layer. Each version is associated with the validity time frame and performance update options, forming a bi-temporal layer by the sequences.

For the evaluation study, three experiments were done:

- Parsing and loading impacts.
- Impact of identifying the proper version.
- Loading the code from the database (or file storage) to the database instance memory.

After all, the scalability of the solutions is discussed.

The first part of the evaluation study emphasizes parsing and version loading impacts. REF solution does not bring any module dealing with the versioning. They are stored externally, outside of the database in the file system. Each version is stored in a separate file, which can be grabbed to start the parsing, checking, and loading process. Although the process is straightforward, many swaps between the database and file storage repository make the solution too demanding. An improved solution (REFopt) creates data version partitions for the execution task. Thus, each version is loaded for the defined code frame only once and executed for each occurrence. Although such structural re-optimization brings benefits at the level of a specific code module, a significant deficiency in the possibility of processing functions and individual versions in a parallelism environment remains. The source files consisting of the function version code are accessible by the database directory object, enhanced by the operating system, and database-granted privileges.

Storing individual version source files in the database layer can bring significant benefits. The code is stored in the database tables using extensive object attribute definitions. First, each code version is directly placed in the database, even in a non-parsed (original) form. The file storage does not need to be handled. Second, the instance of the database system has direct access to the database repository using the background processes, I/O operations are optimized in that manner. The disadvantage of the solution relates to the block-oriented structure. Regardless of the amount of relevant data in the block, the whole block must always be loaded and processed. Additionally, the user must have a suitable method of transforming their original source code into a large object structure. SOL2 enhances the existing SOL1 principles by storing pre-parsed and checked versions as objects. The necessity of parsing and checking is limited since it is done only once during the initial version of the treatment. The loading is more straightforward, although the additional module must be present for converting the pre-parsed version into an executable source. The disadvantage of that approach relates to the necessity of storing the original source code, as well,

so the storage demands are almost twofold. Namely, if the referenced objects are changed, a particular function version is marked invalid and must be compiled before the subsequent usage. It, however, requires the original source code to check the availability of the sources, privileges, proper references, etc.

From the parsing and loading perspectives, SOL3 provides sufficient power. Each version is directly accessible via the unique name created from the original function identifier and version extension. From the calling environment, individual versions are directly accessible through the instance memory structures and system table descriptions. However, for the second evaluation step done in this paper – identifying the proper version, enormous additional demands are present, caused by the necessity to restructuralize user-defined code and apply individual version headers – replace original function names with the version extensions. Consequently, the code definition is changed on the fly, and additional performance optimization is hard to do since the references evolve over time and must be dynamically applied. Note that it is not just about the versions themselves; these can even be extended by several optimization techniques defined over time.

Finally, the proposed solution refers to the temporal paradigm. All the versions are directly parsed and accessible. The original source code does not need to be stored. The dynamic version pointers do direct mapping, so the original names and user code definition remain. Version applicability is done immediately. However, the Hash Plan ID values remain the same, so the execution plans that have already been created remain valid.

Tab. 2 shows the results, expressing the costs. Database execution costs form the metrics covering the database system resources, storage repository, loading necessity, memory consumption, connected background processes, and processing time. For declarative purposes, individual values are expressed in percentages. It is based on calling function F1-F5 10,000 times; the versions are selected randomly. The evaluation was executed ten times, Tab. 2 expresses the average values.

TABLE 2. Parsing and checking impacts.

Parsing and checking impacts	REF	REFopt	SOL1	SOL1opt	SOL2	SOL3	SOL4
Relative costs [%]	100.00%	43.24%	72.77%	39.91%	41.76%	33.12%	27.04%

Whereas the versions to be called are randomly distributed in the calling environment, the worst solution is provided by the REF since it does not handle the parsing, checking, and loading efficiency. REFopt makes partitioning across the versions, and loading of a particular version is done only once for the whole code. By using this optimization technique, total processing costs can be reduced by 56.76%. SOL1 is analogous to the REF but provides an additional security layer by moving the source code to the database repository. The total cost demands are 72.77%, which refers to 27.23%. It is caused by limiting the necessity to contact file repositories outside the database handling. If the optimization by the partitioning is done, total processing demands are 39.91%, reflecting the 7.70% improvement compared to the REFopt (considered as 100%). For the other architectures, version partitioning and categorization to ensure one version is loaded only once are not requested because the versions are pre-parsed or directly accessible by the database system. SOL2 requires 41.76%. However, it is essential to note that compared to optimized version partitioning techniques, such as version categorization is unnecessary. From the point of view of the mentioned statistics, the creation of categories and the grouping of versions are not considered. By using additional evaluations, version partition management requires an additional 5-10% costs (for the evaluated number of versions), depending on the statement or code complexity. With the rise in the number of versions, additional requirements can grow significantly. Solution SOL3 is excellent from the loading perspective. Individual versions are directly available but limited by the different names across the versioning. Based on the defined environment, an additional 22.43% was required to transform the existing source code with the function version name enhancements. It was more pronounced for the SQL language, where it changed the statement code format, from which the execution plan is calculated. Although existing plans can be present in the instance memory Library Cache, the processed Hash Plan ID value differs, so the mapping is not done, and a new execution plan must be built. Furthermore, if any version is changed (even delimited by the performance update), a new execution plan must be created. In SQL, for the defined functions F1-F5, additional demands are 34.12%. The PL/SQL is not so strict. Even if the process plan changes, the function context switches can apply the change more straightforwardly. Precisely, additional demands for PL/SQL are 10.74%. The best solution is provided by the proposed solution SOL4 because the whole management is done on the database level. Each function version is compiled only once, and then directly accessible. The mapping is supervised by the temporal module, which shifts the function dynamically into the particular version route. Furthermore, each version has a pointer to the latest version, so the most optimized solution is always used. The total demands are 27.04%. Compared to the original solution (REF), there is a 72.96% improvement. Reflecting on SOL2's storage of the parsed version in the database, the improvement is 35.25%. Finally, comparing version header naming (SOL3) and our proposed solution, processing costs refer to 81.64% (improvement: 18.36%).

It is worth mentioning that except for the proposed solution SOL4, only the current performance update reflection is stored and thus loaded for each function version. Thus, each version was provided just by one source

code – the latest one (based on the highest sequence number).

The second part of the performance evaluation reflects identifying the proper function version based on the data and validity reference. In this case, REFopt and SOL1opt are irrelevant since they do not provide any additional layer regarding version identification. It does not matter whether this process takes place during version identification or categorization through version partitioning. Based on the evaluation, there is less than a 1% difference between the original and optimized solutions.

REF solution does not store and specify validity ranges, so the code header must always be loaded to identify a suitable version. If it is found, the scanning can be ended, but all the versions need to be evaluated. Furthermore, if it does not identify performance updates, all versions must be scanned in any situation since they can be distributed randomly. Storing versions in the database (SOL1 and SOL2), either in the original or parsed form, makes sense. The data table storing the content is temporal, so the identification is done on the database layer. The bi-temporal architecture allows the management of not only versions themselves but also performance updates for each version. Version identification (SOL3) did not prove itself. Considering the naming and validity, reflection requires access to the temporal layer taking versions and a correlation with the disc storage to locate and transfer particular versions based on their name. Tab. 3 shows the results.

TABLE 3. Identifying proper version.

Identifying proper version	REF	SOL1	SOL2	SOL3	SOL4
Relative costs [%]	100.00%	67.12%	66.59%	93.03%	30.58%

Extracting only current performance updates of the versions into a separate temporal architecture can reduce costs to 25.32%, which refers to the 17.20% improvements.

The last evaluated process is associated with loading the code from the database (or storage repository) to the database instance memory. If the function versions are already available and stored in the database repository, it just needs to be loaded into the instance memory – executable version. To make the evaluation relevant and comparable, instance memory is wholly flushed away before each experiment since the version could be there, so the loading would not be necessary, influencing the results. The flushing uses code unit granularity, so the whole user-specified code is treated as a single unit, encapsulated by the flushing operation from both sides. Tab. 4 shows the results. The REF solution is the most demanding since two steps must be done – loading from the file into the temporary database repository, then loading into the instance memory and consecutive management. SOL1 and SOL2 provide almost the same results. Data are present in the database. The difference between them is based on the size – the parsed version is commonly optimized and, thus, requires a smaller size, so the loading takes a smaller number of blocks and is faster. Based on evaluating the discussed functions, the difference between SOL1 and SOL2 is 10.32%. SOL3 is based on the fact that the versions are present in the system and are referencable by the system tables. Individual versions act as standard conventional functions from the execution point of view because each can act separately, caused by the name resolution. The proposed solution SOL4 combines the benefits of all the above architectures. The only thing that must be done is to load the executable function version content from the database to the instance memory like ordinary functions. From the management's point of view, there is no additional space for optimization. However, reflecting the real applicability, it is worth pointing to the transfer between the database and instance memory. Therefore, in future research, we will emphasize creating function version baselines to define the version priorities because individual versions do not cover the same validity time frame and considered data numbers. Therefore, individual function versions are not executed with the same ratio. The most often used function versions should be preferred to be placed in the memory. Thanks to future evaluation studies and methods to collect function calls, it will be possible to set priorities and transform existing memory structures to serve the executable code more efficiently. This task, however, is not related exclusively to the function versions but can apply to any function type management, even outside of the relational paradigm.

The results reached are shown in Tab. 4.

TABLE 4. Identifying proper version.

Memory loading	REF	SOL1	SOL2	SOL3	SOL4
Relative costs [%]	100.00%	54.91%	49.24%	98.40%	46.06%

A. SUMMARY

The computational evaluation study specified in this section points to three aspects of the temporal function version management and handling. We have been dealing with the parsing, checking, and, generally, compiling process since versions can be stored in the source code format outside the database in ordinary files. This is the most critical activity and takes 50% of the whole process of function management. The second aspect points to logically identifying the correct version, making it accessible for consecutive reference. It takes 32%. The remaining 18% refer to loading the pre-parsed version by making it directly executable. This activity transfers the data from the storage repository (database) into the instance memory. It also includes notifying system tables for the references and object descriptions. Tab. 5 shows the logical costs of the whole process by applying the weights of individual activities. The REF solution obtained the worst results, which does not provide sufficient structures to serve and locate versions. Similarly, SOL3 is not relevant for further analysis and actual use precisely because of the necessity

TABLE 5. Results - summary.

Function version management	REF	REFopt	SOL1	SOL1opt	SOL2	SOL3	SOL4
Costs [%]	100.00%	71.71%	67.75%	51.32%	51.05%	64.04%	31.60%

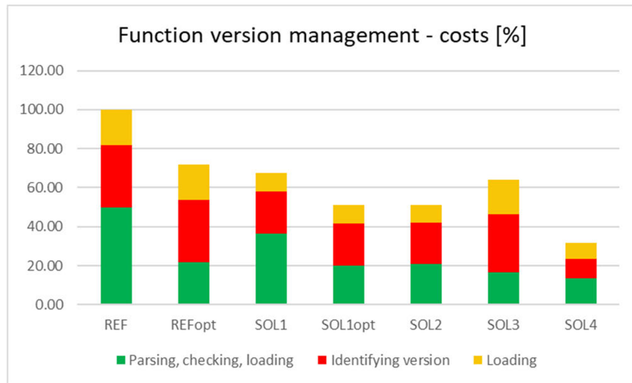


FIGURE 12. Function version management costs.

of changing the execution plan and influencing global system performance. Comparing REF and SOL3, the difference between those two solutions is 10.70%. From a global level, the SOL1opt and SOL2 solutions are very similar, and the code structure retention method has no significant effect. This is, however, caused by the function called pre-partitioning. If it were not there, the total demand of the SOL1 would be 67.75%. The best solution is provided by the model presented in this paper (SOL4). It achieves the best results at all levels. The total processing cost demands are 31.60%. The results are shown in Tab. 5. Graphical representation of the costs is depicted in Fig. 12.

B. SCALABILITY

Based on the presented computational study and results, it is clear that the proposed solution provides significant performance improvements and is powerfully relevant compared to the existing techniques and approaches. However, temporal databases are characterized by the significantly increasing data over time. Compared to conventional databases, in which an update statement replaces the original state, a new tuple is created in temporal databases, and original values remain in the system. Although some purge activities can be identified, such as removing historical data from the system, in a data analytical environment, most of the data should remain accessible. Thus, the number of data to be handled, treated, and stored continuously expands. To serve and declare the flexibility of the proposed solution, this part discusses the scalability of individual solutions. To note, to make the solutions comparable and generally applicable, only optimized solutions for REF and SOL1 will be stated. Sequential version scanning does not make sense because of the strict performance degradation. The initial data set was described at the beginning of this section. Then, two other data sets

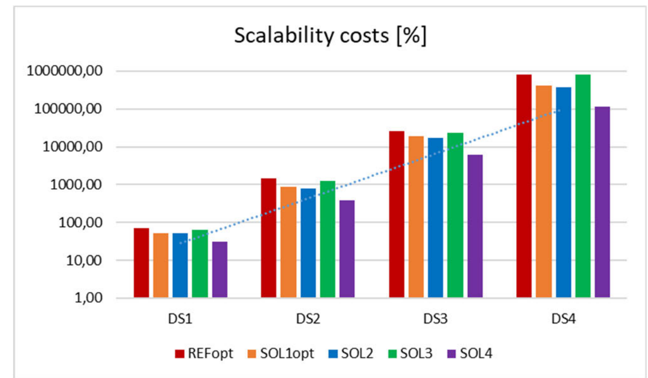


FIGURE 13. Scalability.

with analogous properties were used. The main difference is the amount of temporal data and the related increase in the number of versions. The ratio between the versioning and the amount of data remains preserved. Fig. 13 shows the results.

The first data set takes the original data amount, the second data set is 10 times larger, the third is 100 times larger, and the last and fourth are 1000 times larger. They are denoted by the symbols DS1 to DS4.

VIII. CONCLUSION

Data to be handled by intelligent information systems must be temporally oriented, allowing monitoring of changes over time. Current valid data are still the most critical; however, maintaining historical data offers a vast perspective for analytics and decision-making but also creates future prognoses to adequately react to the changed parameters and properties. The application domain experts must make the decisions based on the complex, accessible, and verifiable data.

In the temporal environment, various architectures have been proposed, varying in terms of processed granularity and data precision, starting with object-oriented data and ending with attribute-delimited temporal data on the second corridor. The intersolution, also discussed in this paper, is delimited by the temporal synchronization groups, by which the stored data do not cover duplicates. Furthermore, it is possible to process multiple attributes as a single unit, optimizing the temporal reference layer.

A natural question regarding function evolution is raised by treating, accessing, and processing temporal databases. This paper discusses the techniques of multiple-function version management and proper mapping based on the validity time frames. The most significant disadvantage of the existing approaches is related to limiting parallelism. Namely, one of the existing solutions takes versioning to the file repository level. Consequently, the proper function version must be dynamically loaded into the system.

This requires not only parsing and the entire checking and loading processes. Above all, only one version of such a function can be in the system at a given time since the entire query evaluation is reduced to sequential processing.

Although various enhancements have been provided in the past, e.g., based on storing versions in the database in the form of large objects or storing pre-parsed function versions, there is still an increasing demand for complex and reliable solutions. Nowadays, it is essential to ensure massive parallelism and support its spread. Our proposed solution emphasizes temporal function version enhancement and mapping through the temporal database to make the function version management applicable, reliable, and scalable. Thanks to that, all versions are accessible in the system and can be directly loaded from the database repository into the memory. All versions are stored in the parsed formats, so the compilation is straightforward. This paper produced several techniques to manage temporal functions, followed by the definition of commands and discussion related to version identification and references in the development system. It goes through the registration of the function and management of individual versions up to the merging process. However, it is not just about versioning. It can be worth optimizing the code specified by the version. Thus, new compilation does not automatically mean releasing a new version, but the existing one can be optimized e.g. using more advanced data structures. The proposed bi-temporal function mapping solution reflects the versioning in the first layer, and the second temporal layer deals with the performance updates.

The core part of the paper covers the definition of temporal functions and existing solutions, which are discussed by pointing to their properties and limitations. The bi-temporal solution takes the function header and maps it to the function body represented by the version codes and interconnected list of changes. The structure is operated by the Function Mapper background process. Another solution is related to the file management holding versions in the LOB repository, followed by the solution storing parsed data in the database and multiple function mapping names. Based on the provided analysis, it is clear that storing the parsed version is inevitable for ensuring the performance of the version mapping. Section V points to the proposed solutions. They primarily handle dynamic version mapping, manage testing and released versions, handle internal version updates, register new versions and map them. The massive parallelism should cover all these activities by enabling aspects of scalability in a complex temporal environment. Section VI drives you through various proposed techniques and enhancements, like command registration, function types, applying PRAGMA keyword, version registration and merging, dynamic recompilation, and performance update registration. All these aspects and characteristics are properly discussed in the section, allowing the creation of complex, robust, and performance-efficient solutions.

The performance evaluation study highlights four architectures managing function versions. Three criteria were considered – costs of the parsing and loading, impact of version identification, and loading the code from the database or file storage, respectively, to the database instance memory. These steps are critical for the process of data retrieval. Based on the experiments associated with flight monitoring, the best solutions were obtained by the SOL4 described by the transformation mapping module introducing a transformation mapping module in a temporal layer. Each version is associated with the validity time frame in the function version header for that solution. The mapping is automatic and operated by the introduced background processes. Overall, it can reduce management, mapping, and loading costs by up to 70%, compared to the reference solution characterized by the temporal versioning located in the source files, followed by the loading and mapping. In this aspect, it is worth mentioning that most of the original solutions differ in versions by name. Thus, the user must ensure proper call of the version.

In addition to the performance, processing costs, and techniques of mapping individual versions, it is essential to emphasize the additional costs associated with managing temporal versions of functions. Namely, version management requires maintaining data structures, mapping, and background processes supervising the whole architecture and processes. Based on the computational study, proposed temporal version management enhancements impact performance by less than 1% in most cases, reflected by the change operations and new version registration.

When dealing with temporal function management and proper version mapping, numerous challenges can be identified associated with the code split, version management, ensuring availability, security, and references using automatic mapping. All these significant difficulties were considered and solved by the proposed solutions. From the performance point of view, there are no significant limitations; the architecture does not require huge additional costs and storage. It reflects less than 1% of additional costs compared to the whole database storage.

The proposed solution does not significantly impact the performance. Based on the computational study, proposed temporal version management enhancements impact performance by less than 1% in most cases. The designed solutions can distinguish between test and release function versions. Individual versions are bi-temporally considered, allowing the change of source code in terms of changed conditions and calculations, but also allowing the optimization of the code itself, with no reflection on the input and provided output values.

As in any practical solution, the proposed methodologies, concepts, and architecture have limitations. It cannot correctly handle non-deterministic functions because such values cannot be pre-calculated and shared across multiple queries and analytical reports. If there is any change that should be applied for multiple versions, they must be handled and recompiled sequentially, causing temporary delays and unavailability. Furthermore, problems are caused by the proposed models, which are characterized by the beginning of the validity of the version only. In that case, determining the proper version is more complicated since there is no strict time frame range. During future development, we will

emphasize the internal transformation to the temporal begin and end date format (BD-ED).

Another limitation that has not been addressed yet relates to the distributed environment. Cloud systems are characterized by high availability, where individual data images, models, and structures are mirrored to multiple cloud repositories and regions by building sophisticated availability domains. To make the system consistent, it is necessary to automatically apply invidiual function versions to all repositories. For now, it is done by the core data synchronization processes, which do not empower the system correctly. Namely, if the function version is not adequately shared across the whole ecosystem, some repositories and locations could call improper versions since a new release is not unavailable there.

In future research, emphasis will be placed on reflecting such functions in the dynamic views, spreading functions across the data distribution architectures, and deeper analysis of the scalable function references. Besides, we will focus on creating function version baselines to optimize the availability of the function versions directly in the instance memory, so the loading between the database storage and instance memory for the execution will not be necessary. This requirement is also reflected in the need for system performance in cloud environments, where a large amount of memory can be provisioned, so the focus should be on the loading and availability.

REFERENCES

[1] S. Chaturvedi and T. Nagpal, "Efficient querying and indexing of moving data objects," in *Proc. Int. Conf. Futuristic Technol. (INCOFT)*, Nov. 2022, pp. 1–6, doi: 10.1109/INCOFT55651.2022.10094348.

[2] N. Mukherjee, S. Chavan, M. Colgan, M. Gleeson, X. He, A. Holloway, J. Kamp, K. Kulkarni, T. Lahiri, J. Loaiza, N. Macnaughton, A. Mullick, S. Muthulingam, V. Raja, and R. Rungta, "Fault-tolerant real-time analytics with distributed Oracle Database in-memory," in *Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE)*, May 2016, pp. 1298–1309, doi: 10.1109/ICDE.2016.7498333.

[3] S. Pendse, V. Krishnaswamy, K. Kulkarni, Y. Li, T. Lahiri, V. Raja, J. Zheng, M. Girkar, and A. Kulkarni, "Oracle Database in-memory on active data guard: Real-time analytics on a standby database," in *Proc. IEEE 36th Int. Conf. Data Eng. (ICDE)*, Apr. 2020, pp. 1570–1578, doi: 10.1109/ICDE48307.2020.00139.

[4] Y.-C. Chen, W.-C. Peng, and S.-Y. Lee, "Mining temporal patterns in time interval-based data," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 12, pp. 3318–3331, Dec. 2015, doi: 10.1109/TKDE.2015.2454515.

[5] P. Terenziani, "Nearly periodic facts in temporal relational databases," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 10, pp. 2822–2826, Oct. 2016, doi: 10.1109/TKDE.2016.2585483.

[6] A. Belussi, E. Bertino, and B. Catania, "An extended algebra for constraint databases," *IEEE Trans. Knowl. Data Eng.*, vol. 10, no. 5, pp. 686–705, Sep. 1998, doi: 10.1109/69.729722.

[7] C.-H. Chen, H. Chou, T.-P. Hong, and Y. Nojima, "Cluster-based membership function acquisition approaches for mining fuzzy temporal association rules," *IEEE Access*, vol. 8, pp. 123996–124006, 2020, doi: 10.1109/ACCESS.2020.3004095.

[8] H. Nindito, A. V. D. Sano, and A. R. Condrobimo, "Comparative study of storing unstructured data type between BasicFile and SecureFile in Oracle Database 12c," in *Proc. Int. Conf. Inf. Manage. Technol. (ICIMTech)*, Nov. 2016, pp. 146–149, doi: 10.1109/ICIMTech.2016.7930319.

[9] S. Arora, "A comparative study on temporal database models: A survey," in *Proc. Int. Symp. Adv. Comput. Commun. (ISACC)*, Sep. 2015, pp. 161–167, doi: 10.1109/ISACC.2015.7377335.

[10] M. Finger, "A logical reconstruction of temporal databases," *J. Log. Comput.*, vol. 10, no. 6, pp. 847–876, Dec. 2000, doi: 10.1093/logcom/10.6.847.

[11] A. Nuijten and P. Barel, *Modern Oracle Database Programming: Level up Your Skill Set to Oracle's Latest and Most Powerful Features in SQL, PL/SQL, and JSON*. New York, NY, USA: Apress, 2023, p. 576.

[12] J. Máté and J. Šafařík, "Transformation of relational databases to transaction-time temporal databases," in *Proc. 2nd Eastern Eur. Regional Conf. Eng. Comput. Based Syst.*, Sep. 2011, pp. 27–34, doi: 10.1109/ECBS-EERC.2011.14.

[13] A. Beiraimi, K. Pu, and Y. Zhu, "Towards optimal snapshot materialization to support large query workload for append-only temporal databases," in *Proc. IEEE Int. Congr. Big Data*, Jul. 2018, pp. 268–271, doi: 10.1109/BIGDATACONGRESS.2018.00048.

[14] M. Kvet, "Enhanced data locking to serve ACID transaction properties in the Oracle Database," in *Proc. 34th Conf. Open Innov. Assoc. (FRUCT)*, Nov. 2023, pp. 73–80, doi: 10.23919/fruct60429.2023.10328165.

[15] M. Kvet and J. Papan, "The complexity of the data retrieval process using the proposed index extension," *IEEE Access*, vol. 10, pp. 46187–46213, 2022, doi: 10.1109/ACCESS.2022.3170711.

[16] M. Yu, C. Chai, and G. Yu, "A tree-based indexing approach for diverse textual similarity search," *IEEE Access*, vol. 9, pp. 8866–8876, 2021, doi: 10.1109/ACCESS.2020.3022057.

[17] *Developing Robust Date and Time Oriented Applications in Oracle Cloud: A Comprehensive Guide To Efficient Date and Time Management in Oracle Cloud 1, Kvet, Michal—Amazon.com*. Accessed: Feb. 6, 2024. [Online]. Available: https://www.amazon.com/Developing-Robust-Oriented-Applications-Oracle-ebook/dp/B0BZJG8G7Q/ref=sr_1_1?crid=1JD46TW1GIW0S&keywords=Developing+Robust+Date+and+Time+Oriented+Applications+in+Oracle+Cloud&qid=1707159540&sprefix=developing+robust+date+and+time+oriented+applications+in+oracle+cloud%2Caps%2C243&sr=8-1

[18] N. Mahmood, S. M. A. Burney, S. A. Ali, K. Rizwan, and S. A. K. Bari, "Fuzzy-temporal database ontology and relational database model," in *Proc. 9th Int. Conf. Fuzzy Syst. Knowl. Discovery*, May 2012, pp. 573–577, doi: 10.1109/FSKD.2012.6233841.

[19] M. Hudec, "Fuzzy data in traditional relational databases," in *Proc. 12th Symp. Neural Netw. Appl. Electr. Eng. (NEUREL)*, Nov. 2014, pp. 195–200, doi: 10.1109/NEUREL.2014.7011504.

[20] F. Kong, B. Jiang, G. Yue, and J. Wang, "Deep data imputation for UAV low-altitude sensing considering spatial–temporal interaction," *IEEE Sensors J.*, vol. 24, no. 6, pp. 8170–8183, Mar. 2024, doi: 10.1109/JSEN.2024.3354330.

[21] H. Yu, X. Zhang, Y. Wang, Q. Huang, and B. Yin, "Fine-grained accident detection: Database and algorithm," *IEEE Trans. Image Process.*, vol. 33, pp. 1059–1069, 2024, doi: 10.1109/tip.2024.3355812.

[22] M. Zhu and T. Risch, "Querying combined cloud-based and relational databases," in *Proc. Int. Conf. Cloud Service Comput.*, Dec. 2011, pp. 330–335, doi: 10.1109/CSC.2011.6138543.

[23] V. K. Myalapalli and B. L. R. Teja, "High performance PL/SQL programming," in *Proc. Int. Conf. Pervasive Comput. (ICPC)*, Jan. 2015, pp. 1–5, doi: 10.1109/PERVASIVE.2015.7087001.

[24] M. Adnan and R. Alhajj, "A bounded and adaptive memory-based approach to mine frequent patterns from very large databases," *IEEE Trans. Syst., Man, Cybern., B, Cybern.*, vol. 41, no. 1, pp. 154–172, Feb. 2011, doi: 10.1109/TSMCB.2010.2048900.

[25] K. J. Upadhya, A. Paleja, M. Geetha, B. D. Rao, and M. S. Chhabra, "Finding partial periodic and rare periodic patterns in temporal databases," *IEEE Access*, vol. 11, pp. 92242–92257, 2023, doi: 10.1109/ACCESS.2023.3308820.

[26] N. T. Nguyen, "Consensus-based timestamps in distributed temporal databases," *Comput. J.*, vol. 44, no. 5, pp. 398–409, Jan. 2001, doi: 10.1093/comjnl/44.5.398.

[27] *Performance Analysis of PL/SQL Query Optimization Techniques|IEEE Conference Publication|IEEE Xplore*. Accessed: Jan. 31, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/8991362

[28] A. Vaisman and E. Zimányi, *Data Warehouse Systems: Design and Implementation*. New York, NY, USA: Springer, 2014, p. 625.

[29] A. de Mauro, F. Marzoni, and A. Walter, *Data Analytics Made Easy: Use Machine Learning and Data Storytelling in Your Work Without Writing Any Code*. U.K.: Packt Publishing, 2021.

[30] R. Tian, H. Zhai, W. Zhang, F. Wang, and Y. Guan, "A survey of spatio-temporal big data indexing methods in distributed environment," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 15, pp. 4132–4155, 2022, doi: 10.1109/JSTARS.2022.3175657.

[31] R. Ma, X. Han, L. Yan, N. Khan, and Z. Ma, "Modeling and querying temporal RDF knowledge graphs with relational databases," *J. Intell. Inf. Syst.*, vol. 61, no. 2, pp. 569–609, Oct. 2023.

[32] B. El Idrissi, S. Baïna, A. Mamouny, and M. Elmaallam, "RDF/OWL storage and management in relational database management systems: A comparative study," *J. King Saud Univ. Comput. Inf. Sci.*, vol. 34, no. 9, pp. 7604–7620, Oct. 2022.

[33] Q. H. Le and M. Diaz, *Developing Modern Database Applications With PostgreSQL: Use the Highly Available and Object-Relational PostgreSQL to Build Scalable and Reliable Apps*. U.K.: Packt Publishing, 2021, p. 417.

[34] P. R. G. Cordeiro, G. D. C. Cavalcanti, and R. M. O. Cruz, "Dynamic ensemble algorithm post-selection using hardness-aware Oracle," *IEEE Access*, vol. 11, pp. 86056–86070, 2023, doi: 10.1109/ACCESS.2023.3304912.

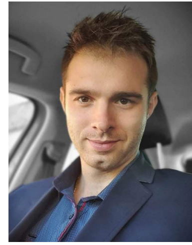
[35] V. K. Myalapalli and P. R. Savarapu, "High performance SQL," in *Proc. Annu. IEEE India Conf. (INDICON)*, Dec. 2014, pp. 1–6, doi: 10.1109/INDICON.2014.7030467.

[36] N. Wang, A. Kamali, V. Kantere, C. Zuzate, V. Corvinelli, B. Frendo, and S. Donoghue, "A hybrid cost model for evaluating query execution plans," in *Proc. IEEE 6th Int. Conf. Artif. Intell. Knowl. Eng. (AIKE)*, Sep. 2023, pp. 133–138, doi: 10.1109/aike59827.2023.00030.

[37] Z. Ling, L. Qi, Z. Qianyuan, and C. Wei, "The study of adjustment and optimization of Oracle Database in information system," in *Proc. Int. Conf. Comput. Inf. Sci.*, Jun. 2013, pp. 442–445, doi: 10.1109/ICCIS.2013.123.

[38] A. Badia, *SQL for Data Science: Data Cleaning, Wrangling and Analytics With Relational Databases*. New York, NY, USA: Springer, 2020.

MICHAL KVET (Member, IEEE) became an Associate Professor in applied informatics with the Faculty of Management Science and Informatics, University of Žilina, Slovakia, in 2020. He is currently a recognized researcher, a conference speaker, and an Oracle ACE Alumn. He is the author of several textbooks and monography in temporal database processing. He is the author of more than 70 scientific articles indexed in IEEE-Xplore, Scopus, or WOS. He is certified for SQL, PL/SQL, analytics, and cloud databases. His research is devoted to temporal databases, indexing, performance, analytics, and cloud computing. He strongly participates with Oracle Academy and he is a part of multiple Erasmus+ projects. Besides, he is a Consortium Leader of the Erasmus+ project dealing with environmental analytics. He also organizes multiple database workshops annually.



JOZEF PAPAN received the Ph.D. and Doctorate degrees in applied informatics from the Faculty of Management Science and Informatics, University of Žilina, Slovakia, in 2015 and 2020, respectively. He is currently the Head of the IP Fast Reroute Research Team, the Director of the Fortinet Network Security Academy, and a member of Cisco Academy with the Faculty of Management Science and Informatics. He is also the Teacher of the following subjects: Securing Networks with Fortinet (Fortinet Academy), Principles of ICS (Cisco), and Network Architectures (Linux + Networks). He is the author or coauthor of more than 30 scientific papers published in scientific journals and presented at international conferences. His research interests include IP fast reroute, fault-tolerance, protocols and services in IP networks, WSN, the IoT, modeling and simulation of computer networks, smart sensors, wireless technology, portable devices, technical cybernetics, and cloud computing.

MARTINA HRÍNOVÁ DURNEKOVÁ is currently pursuing the Ph.D. degree in applied informatics with the Faculty of Management Science and Informatics, University of Žilina. Her research interests include data analysis, data warehouses, architectures and performance impacts, and scalability and cloud technologies.