IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 5 August 2025, accepted 14 August 2025, date of publication 18 August 2025, date of current version 25 August 2025.

*Digital Object Identifier 10.1109/ACCESS.2025.3599972*

RESEARCH ARTICLE

# A Scalable Approach to Historical Data Management via Optimized Transaction Logs

**MICHAL KVET[1], (Senior Member, IEEE), AND JARMILA ŠKRINÁROVÁ[2]**
[1]University of Žilina, 010 26 Žilina, Slovakia
[2]Matej Bel University, 974 01 Banská Bystrica, 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** Temporal databases play a crucial role in tracking the evolution of data over time by preserving historical states through valid time intervals instead of overwriting data. This approach enables robust data analysis, forecasting, and auditing while enforcing consistency and preventing conflicting entries. Traditional transactional systems complement this by recording the time context of changes in transaction logs, which are essential for ensuring data integrity, recovery, and auditability. However, conventional log management, relying on sequential scanning, imposes performance and scalability limitations. This paper introduces a novel, efficient solution for managing transaction logs through an integrated data mapping layer. The proposed structure stores direct references to logged data within the database, supported by five optimized indexes—including object definitions and four temporal B+trees—to track validity, load time, and transaction lifecycle. This block-oriented design enables dynamic access, supports prohibited history handling, and eliminates the need for separate temporal layers or data migrations. Performance evaluations demonstrate significant gains in processing speed and system scalability compared to traditional approaches. However, limitations persist, including the growing footprint of non-indexed log metadata, challenges in data recoverability, incompatibility with standard table-optimization techniques, static log block sizes, and potential bottlenecks during high-frequency change periods.

**INDEX TERMS** Temporal databases, transaction logging, performance optimization, archiving.

## I. INTRODUCTION

A relational database is a type of database that stores and organizes data in a structured way using tables. These tables are made up of rows (records) and columns (attributes). Each table represents a specific type of data. Relationships between these tables are established through keys, primary key or candidate primary key set member (unique constraint) and foreign keys for the child records [1]. The attributes form the main elements and characteristics of the table. Each attribute is delimited by the data type and other constraints can be present, as well. It uses *Entity-Relationship (ER) Diagram*, dealing with the entities, attributes and relationships. To interact with the relational database, SQL (Structured Query Language). From this paper point of view, the most significant category, data modeling language (DML) is important by manipulating the database content using *Insert*, *Update*, *Delete* and *Select*operations.

### A. HISTORY AND THE KEY CONCEPTS OF THE RELATIONAL PARADIGM

Before the relational model was introduced, databases were typically based on *hierarchical model* (used data stored in a tree structures, like IMS IBM's Information Management System) or *network models* (data were represented as graph with the relationships, often requiring manual reference handling), which were much more rigid and complex to manage. These models required navigating through relationships between data using pointers or tree structures, which was difficult for users and developers [4], [5], [6].

The associate editor coordinating the review of this manuscript and approving it for publication was Ayman El-Baz.

The *relational model* was introduced in 1970 by Edgar F. Codd, a researcher at IBM. His paper, "*A Relational Model of Data for Large Shared Data Banks*" formed the core element of the relational paradigm by organizing data into tables consisting of the tuples (rows) and attributes (columns) [7]. The whole concept was based on set processing - operations on data were based on set theory, allowing for querying and manipulation of data using mathematical operations. The data manipulation is done by *relational algebra*operations. Codd's model was revolutionary because it was based on formal mathematical principles, which allowed for a flexible and efficient way of managing large amounts of data. It was based on the *data normalization* ensuring data integrity, consistency by limiting any anomaly [8], [9].

Following Codd's work, several key developments and implementations of relational databases began to emerge – System R by IBM (1974), Relational Software (1977) later recognized as Oracle Corporation (1979) and Ingres (1977) developed at the University of California, Berkeley.

During the 1980s, SQL (Structured Query Language) became the standard query language for relational databases, found on ANSI SQL standard approved in 1986, generally supporting SQL across different relational databases.

In the 90-ties of the 20th century, relational databases saw widespread adoption across businesses of all sizes. Several commercial database management systems emerged and evolved to meet the growing demand, Microsoft SQL Server (1989), MySQL (1995) and PostgreSQL (1996). MySQL is a bit specific, since it was released as another open-source relational database, gaining popularity for its speed and reliability, especially in web development [10].

### B. DATA MANAGEMENT AND SCALABILITY ASPECT

As data storage needs grew, an intensive focus on the scalability and reliability was introduced. Many challenges were solved by handling semi-structured or non-structured databases after introducing *NoSQL database* types [11], [12]. Another stream was defined by the cloud databases [13], [14], [15], in which the Oracle Database can be considered as a leader due to the autonomous database definition managing transactional, as well as analytical-oriented database workload.

Despite the rise of NoSQL and cloud databases, relational databases remain a dominant force in many industries due to their robustness, maturity, and established standards.

To ensure the data reliability, consistency and error-prone, data need to be normalized in the relational schema.

Fig. 1 shows the principle of the data flow by pointing to the data retrieval using indexes and advanced storage features. To optimize the performance, those main aspects should be considered, treated and handled:

- database system architecture,
- data modeling,
- indexing enhancements,
- partitioning and data distribution.

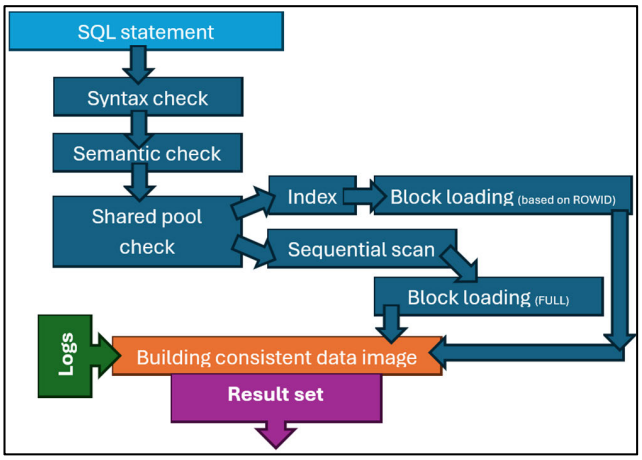



**FIGURE 1.** Query processing diagram.

### C. TRANSACTION SUPPORT

The relational paradigm of data processing is directly associated with the transaction management and support. A *database transaction* is a sequence of database operations (such as insert, update, delete, or select) that are executed as a single unit. Transactions ensure that the database remains in a consistent state even in the face of errors, system failures, or crashes. A database transaction refers to the following properties with the ACID acronym [2], [7], [25]:

- *atomicity* (A) – atomicity ensures that the transaction either completes entirely (all operations succeed) or not at all (if any operation fails, the transaction is rolled back) by applying all-or-nothing approach.
- *consistency* (C) – consistency ensures that all the constraints, either defined on the data model layer, integrity or code are passed before the transaction is approached. In other words, transaction shifts the database from one consistent state to another, which is also consistent. However, during the transaction run, some constraints can be violated, like referential integrity.
- *isolation* (I) – isolation enables huge concurrency, meaning, that transactions are isolated during the run. The applied change is generally available after reaching transaction end (Commit). This prevents issues like dirty reads, non-repeatable reads, or phantom reads.
- *durability* (D) – durability ensures that once a transaction is committed, its effects are permanent and will survive any system crashes. The changes made by the transaction are written to disk and will not be lost. Key aspects of the durability are persistence of changes, data integrity and commit log.

  In principle, there are two options (principles) for ensuring durability:
  - *By using Commit Log – Write-Ahead log rule* making all transaction changes logged before the data are written to the database. Thus, the log is stored in the disc storage, even for the approved transaction. Therefore, even if the data themselves are not stored permanently

in the database, by using *Commit Log*, it is possible to reconstruct the state after the failure. *Write-Ahead log*means that the core perspective ensuring durability and integrity is just the transaction log, not the data stored in the database. Such a concept is often used, since it provides a robust performance compared to the direct data management in the block-oriented shape.

- *By using data flushing to the disc* – data changes must be persisted to disk before the transaction can be considered fully committed. Regardless of data caching, it has a strong impact on the I/O operations between the instance memory and physical database storage.

Another aspect covered by the transaction logging is isolation. In a multi-user database environment, multiple transactions can occur simultaneously. Without proper isolation, transactions may affect each other in ways that could lead to incorrect or inconsistent results. For example, if two transactions are updating the same data at the same time, they could overwrite each other's changes or cause other anomalies.

To address this, isolation levels define the extent to which one transaction is isolated from other concurrent transactions. The level of isolation balances performance (how many concurrent operations can be processed) against consistency (how much control is applied to prevent unwanted effects from concurrency).

The SQL standard defines four isolation levels, each with different trade-offs between concurrency and consistency [7], [26]:

- *Read uncommitted* - Transactions are allowed to read data that have been modified but not yet committed by other transactions. This is called a *dirty read*. A transaction could read data that is later rolled back, leading to inconsistency.
- *Read committed* - A transaction can only read data that have been committed by other transactions. It allows non-repeatable reads, where a transaction might read a value, but that value might be changed if the same transaction reads it again (due to another transaction committing changes).
- *Repeatable read* - Ensuring that if a transaction reads a value, it will get the same value every time it reads that piece of data within the transaction (no non-repeatable reads). It allows *phantom reads*, where a transaction reads a set of records that match a condition, but other transactions can insert or delete records that would match the condition, causing the result set to change.
- *Serializable* – It guarantees the highest level of isolation by ensuring that the results of concurrent transactions are equivalent to running them serially (one after the other). The database ensures that concurrent transactions behave as if they were executed in some serial order.

Tab. 1 shows the summary of the isolation level properties.

In this section, we have been dealing with the core concept of relational database technology, by pointing to the integrity and transaction support, which are the key concepts used in our research.

**TABLE 1.** Isolation level summary.

| Isolation Level | Dirty Reads | Non-repeatable Reads | Phantom Reads | Concurrency |
|---|---|---|---|---|
| **Read Uncommitted** | Allowed | Allowed | Allowed | Highest |
| **Read Committed** | Not Allowed | Allowed | Allowed | High |
| **Repeatable Read** | Not Allowed | Not Allowed | Allowed | Medium |
| **Serializable** | Not Allowed | Not Allowed | Not Allowed | Lowest |

*Transaction logging* is a critical process in the databases to ensure recoverability, integrity and isolation. Among that, they are used to serve these purposes – audit trails, consistency, replication and synchronization, data diagnostics, backup, restore and legal compliance.

*Transaction log* stores original and new state by forming change vectors. Log structures are operated by the instance background processes. Thus, it is possible to form temporal data based on the transaction logs. It can be done by scanning log change vectors to get the historical states.

In this paper, transaction log principles are investigated by focusing on the ability to map them into the temporal (historical) approach. It uses archiving to preserve all logs in a specific repository.

The main contribution of this paper relates to the own layer mapping object changes to the change vectors. Thanks to that, instead of necessity to sequentially scan the transaction logs, direct approach, similar to the index data address, can be used. To do that, new processes, memory and database structures are introduced.

Non-indexed metadata refer to log-related metadata that are stored but not searchable via an index (or not indexed for performance or cost reasons). This means you can't quickly query/filter/search based on that data unless you're scanning logs directly in a sequential way. The physical structure is made based on the transaction reference to serve a consistency aspect of the transaction meaning, that the data are sorted based on the transaction, not the object references.

A typical example is data from road infrastructure, where we monitor the current state, based on which the traffic is managed. Changes are captured in transaction logs, but only the current image of the road infrastructure is directly visible. Thanks to the solution we have proposed, it is possible to evaluate changes over time, monitor historical development

and thus support better data-oriented decision-making. Naturally, the proposed concept can also be used in other areas, such as banking, intelligent systems, sensory data processing or medicine. In any area where it may be advantageous to monitor changes over time, but without the need to model the temporal system explicitly.

Performance evaluation study is proposed to declare usability, applicability and power. It is implemented using the Oracle Database, 23ai version.

The paper is organized as follows: Section II deals with the key starting points, referenced solutions and relation work to serve the technical background. Section III summarizes temporal database systems and approaches. It aims to identify functionalities, which need to be implemented on the log data layer. It handles temporal granularity, temporal models, transaction logs, Archive log mode and Flashback. Section IV forms the contribution of the paper by discussing the proposed solution, which is then evaluated in section V, followed by the summary and conclusions.

## II. REFERENCED SOLUTIONS AND RELATED WORK

Relational databases are based on storing current valid data [7]. It means, that the core database structure points only to the data, which represent valid facts at the moment. As a result, when an update occurs, the original data are overwritten, and when a deletion is performed, the corresponding storage is freed—effectively erasing any record of previous states. The system itself offers no built-in mechanism to retain or reconstruct historical information. To sharpen that, the conventional paradigm of the relational database is limited to non-evolving states or management of data. This poses a significant limitation in scenarios where data evolution over time is meaningful. Since relational databases discard previous values during standard operations, they are unsuitable for applications requiring auditability, traceability, or historical analysis.

The strategies to enhance conventional definition by the temporal aspects include using validity columns by tracking the time period during which a record is valid – *valid_from* and *valid_to* [26]. To make it reliable, it is necessary to define representation of the time interval and a way to model unbounded validity. Typically, currently valid states cannot be limited by the validity of the right site of the frame explicitly, they are defined as "*until further notice*", meaning that the state is valid until the next change [27], [28]. When, and whether, this will happen at all, it is unknown at the moment.

Another approach extending conventional paradigm is based on using soft deletes by implementing status of the row – *is_active*, *is_deleteted*, etc. It can hold boolean value or temporal reference – date and time based on the desired value precision [28], [29]. While this technique allows the database to preserve records that are logically deleted or inactive, it introduces several significant limitations, like lack of standardization, data semantics, increased query complexity, scalability issues, but mostly significant rise of the data storage demands.

Previous approaches can be enhanced by archiving old data into a specific repository. Outdated records are moved to the archive table with the same structure. This allows to keep only valid data in the main tables. It was primarily based on limiting amount of data in the main database by enhancing performance, since sequential scanning takes all associated data blocks. Such a shift can be typically done through automated processes like scheduled jobs or triggers.

Time reference checking is based on storing timestamp references expressing create and last update timepoints – *created_at*, *updated_at*. Such an approach cannot refer to the temporal definition, it is used just for the audit of the last executed operation.

*Data expiration* [28], [29], [30] approach was introduced soon after the release of the conventional paradigm by enabling historical data management using triggers. Namely, if there was an attempt to update a row, the original state was shifted into a specific repository. It aims to clean the database by removing expired data. Expiration could be set on database, table or object level. It takes various benefits, like historical data modeling, flexibility and data integrity. Flexibility of that approach is delimited by the ability to reference historical data for the auditing and consecutive reference. On the other hand, it brings various drawbacks. First, current valid data are separated from the historical states. By considering state evolution, both structures must be interconnected using UNION ALL function operation. Secondly, if the structure of the table is changed (e.g. by adding new attribute, changing constraint or even dropping an attribute), particular change must be applied to both structures, which can cause problems with the historical data consistency. The main drawback of this approach is, however, delimited by storage efficiency. Namely, the triggers copied the whole states irrespective of the real change. Thus, many duplicate values were present. Identification of the real change was complicated. Fig. 2 shows the principle of preserving historical data. Later, *NULL* values were considered as a suitable placeholder for expressing non-changed attribute value. However, *NULL* value itself can be a valid value for the attribute and can have a specific meaning, so it is not suitable for modeling attributes, which values were not changed.

Original data expiration approach brought one specific limitation associated with the reliability and security. Specifically, there may be certain attributes for which historical data cannot be stored, either due to privacy concerns or data protection requirements. To address this issue, two primary solutions have been implemented.

First, the sensitive attribute set can be excluded entirely from the archive table. While this protects the data, it requires maintaining two different versions of the data model—one for current operational use and another for historical storage—resulting in increased complexity and potential inconsistency.

Second, the values of the protected attributes can either be omitted (represented as NULL in the historical records) or anonymized before storage. Although this preserves the structure of the historical data, it compromises the complete-

ness and interpretability of the archived records, reducing their utility for auditing, debugging, or analytical purposes.

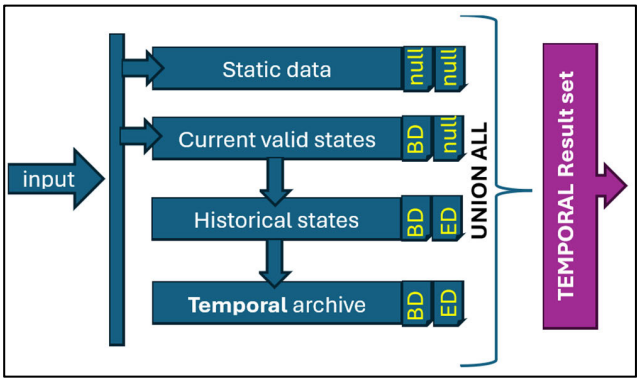



**FIGURE 2.** Model of the various temporal aspect management delimited by the validity frames.

Enhanced model [31] reflecting protected attribute set is depicted in Fig. 3.

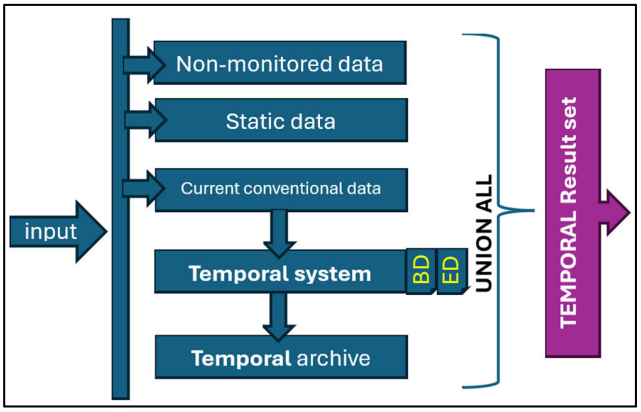



**FIGURE 3.** Enhanced model dealing static, non-monitored data and temporal ecosystem represented by the conventional data layer storing current valid data and temporal layer monitoring changes over time.

## III. TEMPORAL DATABASE MODELING

Temporal databases refer to the tracking of the historical states by allowing to monitor tuple changes over time. It uses sophisticated methods to identify, capture and extract changes of the object states. Temporal databases are designed to handle time-sensitive data, making it possible to store not only the current state of information but also its historical states, and even future states [29].

Temporal databases are basically characterized by two temporal dimensions – transaction time (*TT*) referencing the time, when the row was inserted, updated or deleted, respectively. It can be enhanced by distinguishing between the timestamp of the operation itself, or transaction approval. This can be critical especially in systems with long-lasting transactions and massive parallelism. The second temporal dimension relates to the valid time (*VT*) expressing time period during which a fact is true in the real world, independent of when it was entered or modified in the database [30].

### A. TEMPORAL GRANULARITY

Over the decades, various granularities and precision levels of the processing were handled, starting from the object level granularity up to synchronization groups. In this context, *object-level temporal system* expands the definition of the conventional primary key by adding temporal elements. Thus, each state is primarily bordered by the begin (*BD*) and end (*ED*) point of the validity, formed as direct elements of the primary key. The main disadvantage lies in the necessity to store whole object anytime any attribute value is changed.

*Attribute-oriented granularity* uses similar approach by framing individual attributes into temporal frames. Only changed attributes are referenced by ensuring the stability and efficiency of the processing. On the other hand, querying and data retrieval are far more complicated, since the object state needs to be composed from the individual attributes. Furthermore, if some attributes correlate and are always changed together at the same time, they need to be processed separately, which naturally limits processing efficiency and adds additional demands and costs [30], [31].

*Synchronization group management* was introduced in 2017 by using AI techniques to identify data groups, which are then handled as a single unit, instead of separate processing of individual attributes. Thanks to that, data management and storage reflection can be optimized. Principles can be found in [31], Fig. 4 shows the architecture. Data layer of the synchronization group composition is depicted in Fig. 5.

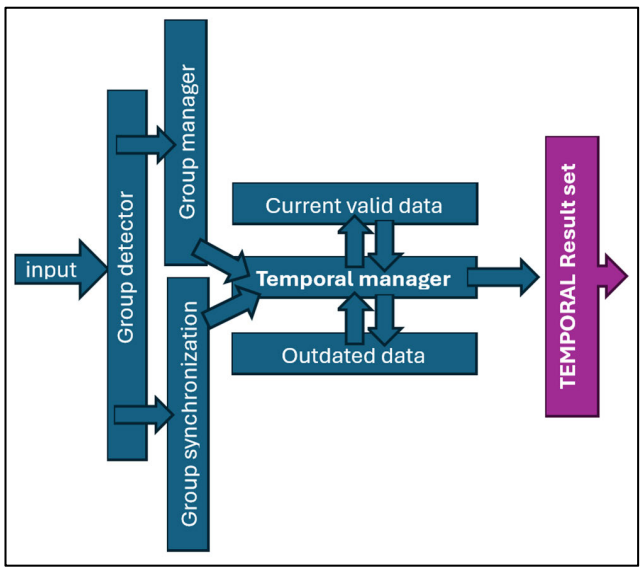



**FIGURE 4.** Model of the synchronization group definition, handling, monitoring and refreshing, secured by the temporal manager splitting the data content into current data and outdated.

The data model of the synchronization group management is in Fig. 5. *Group_def* attribute is a root of the ISA hierarchy, which can reflect individual attribute itself, but also can be

composed by the set of attributes. From that perspective, based on [29], [30], and [31], existing groups can be merged or divided dynamically over time. Thus, the synchronization group definition is enhanced by the object-level temporal model.

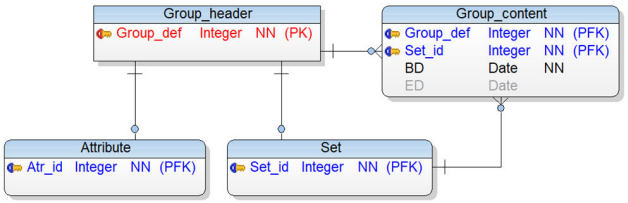



**FIGURE 5. Temporal group management – data model defining synchronization group composition, which can hold one attribute or can be formed by multiple existing synchronization groups using merge operation.**

Temporal granularity refers to the level of detail or precision in the measurement or representation of time in a given context. It essentially determines how finely or coarsely time is divided. The concept is often used in various fields such as data analysis, forecasting, scheduling, and system design. Depending on the purpose, temporal granularity can range from very broad to very detailed. For example:

- *High temporal granularity*: Measuring time in small units like, milliseconds, microseconds or even nanoseconds. This is often used in fields such as high-frequency trading, real-time monitoring systems, or scientific experiments that require precise timing.
- *Low temporal granularity*: Measuring time in larger units such as minutes, hours or days. This would be used for long-term trends, economic forecasting, or historical analysis.

### B. TEMPORAL MODELS

Bi-temporal approach [29], [30], [31] uses *TT* and VT as an extended object identifier. Thanks to that, existing data tuples can be updated expressing the data correction. Furthermore, it can reliably manage and identify delays in the processing (reflecting the positive difference between *TT* and *VT*). A simplification of the bi-temporal model uses only one temporal dimension by forming uni-temporal approach. Either VT or *TT* is used, but not both. It is simpler for management (identifying valid state, time frame overlaps, etc.), but data corrections cannot be handled.

Generally, multi-temporal approach can be used.

When dealing with the temporal modeling extension, it is worth mentioning necessity to synchronize temporal references across multiple time zones or server client. When using clouds and images in multiple repositories, the aspect of data correctness and reliable temporal mapping is even more important.

There are several advantages of the temporal data management summarized in the following list:

- data management covering full history,
- audit data enabling,
- version control systems,
- powerful data analysis – advanced reporting, trend analysis and forecasting.

On the other hand, temporal databases face the following challenges [31], [32], [33]:

- increased system complexity,
- additional storage demands,
- performance degraded by the triggers and jobs managing historical data,
- purging historical data (how long the historical data should be preserved reflecting the balance between the storage demands and information value of too historical data),
- data security (protecting data changes using transaction logs).

### C. USING TRANSACTION LOGS

The primary context of the transaction log is to protect the database and ensure the ACID properties of the transaction. *Transaction log* is operated by the *Log Writer* background process. It consists of the entry header by pointing to the transaction reference, temporal point to the timeline (*System Change Number (SCN)* or timestamp respectively). Optionally, individual logs are chained, so the reference to the previous record can be stored. Transaction information consists of the referenced *operation* (Insert, Update, Delete, Select), *change vector* is part of the *UNDO* and *REDO* segment information. Thus, all the operations modifying database content are logged. Even *Select statement* can cause data change, since the locks for the rows can be physically released (they are released at the end of the transaction only logically).

*UNDO* and REDO structures form the change vector by building before and after images. *Before image* states of the data before the transaction was applied (for *UNDO* operations in case of applying *Rollback*), while *After image* preserves the states of the data after the operation has been applied (used for the *REDO* operations during recovery). *UNDO* data are stored in the database repository covered by the *UNDO tablespace*. Thanks to that, even after the system crashes, particular data are physically stored and available, so the previously running transactions can be rollbacked. Vice versa, *REDO*s are temporarily stored in the instance memory. Just before the transaction reaches the Commit, *REDO* content is moved to the database to pass the durability aspect of the transaction. Therefore, the data themselves in form of the data blocks are not moved, only particular *REDO*, consequencing in high performance and throughput. Besides, *Log Writer* copies the *REDO log* content dynamically based on the following conditions, resulting in simplifying and accelerating the transaction completion process (Commit):

- when the REDO log buffer is one-third full,
- every 3 seconds (timeout),
- before DBWR process writes dirty buffers,
- when a log switch occurs,

• when a user process performs certain operations – DDL or archiving,
• at Commit.

Among that, database transaction log points to many other referenced housekeeping operations used for the audits, recovery, replication, like checkpoint. Naturally, *Commit*, *Rollback* and *Savepoints* are stored, as well.

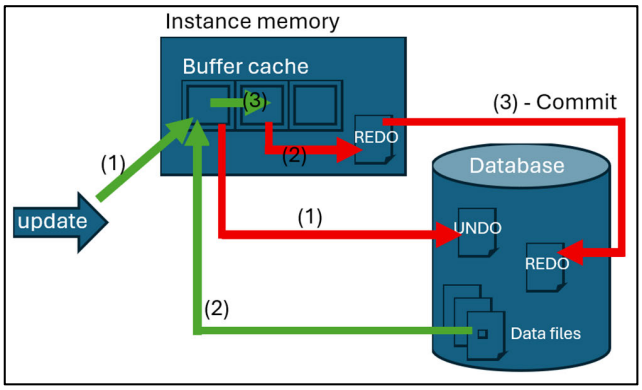



**FIGURE 6. Log management – UNDO/REDO location and movement during the transaction.**

Fig. 6 shows the architecture of the database system from the data flow and log point of view. Green line refers to the data flow by highlighting database as a physical storage repository and *Buffer cache* for storing data in an instance memory. The data themselves are block oriented, thus the read and write operations must take whole blocks, even if the data portion is tiny. Similarly, sequential data scanning requires block loading from the database to the instance memory for the processing and evaluation. If there is a change on the data, it is done in the memory, so the database layer itself does not need to store current valid data, forming the data versioning. On the other hand, transaction ACID properties ensure the ability to recover the data in case of any error, which is done by the transaction logs. Red line shows the log management by pointing to the transaction logs. *Online REDO log* is stored in the instance memory – *Log Buffer*, while *UNDO* is stored in the database. Transaction end reflects the data movement from the instance memory to the database. Thus, the data stored in the database can be outdated, however, by applying transaction logs, current versions can be calculated and provided.

From the ACID point of view, *UNDO* data are necessary only for active transactions to be able to revert them. After reaching Commit, particular data are marked as inactive and can be overwritten. But the database attempts to preserve those logs as long as possible, subject to the disk capacity. If it the repository was filled, further transactions would not be able to be processed. Without keeping logs, it would be impossible to restore a transaction in the event of an error or rejection of the transaction itself.

*REDO* log is necessary not only for the active transactions, but serve as a key element for reconstructing database in case of failure. Thus, if the data images are not stored in the database, only in the instance memory, particular *REDO* logs must be preserved. The operation moving data from the instance memory *Buffer cache* to the database is called *Checkpoint*. After a Checkpoint, the database is in a consistent state that can be used to recover the system to that exact point in case of failure. In current databases, partial checkpoint are primarily used, compared to the full checkpoints used in the past [7], [8], [9].

### D. SNAPSHOT TOO OLD PROBLEM

In the databases, *Snapshot too old* error can occur when a query tries to access data that have been overwritten in the *UNDO* segments (used to store previous versions of data for rollback purposes). When a Select statement is executed, database provides the consistent version of the data as it was at the start of the transaction or operation, respectively. To do so, *UNDO* segments are used. However, if the *UNDO* segments that store older versions of data are overwritten (because the *UNDO* space is full and new data is being written to it), the query might encounter the *ORA-01555: snapshot too old error. ORA-01555: snapshot too old error* [32].It *can* happen for long-running queries, improperly set UNDO tablespace capacity or high DML activity, when lot of change operations (Insert, Update, Delete) are present.

The solutions limiting the *Snapshot too old* exception include [33]:

- increasing *UNDO* tablespace size,
- increasing *UNDO_RETENTION* value to hold inactive UNDOs to ensure data are kept long enough for long-running queries,
- tuning SQL statements, breaking them to smaller portions,
- increasing frequency of checkpoints making the database content consistent more often,
- using *Flashback technology* by enhancing data transaction log structures.

### E. ENABLING ARCHIVE LOG MODE

To record transaction logs for a longer time, *Archive log mode*of the database can be enabled. This is primarily important for data recovery and for maintaining a backup strategy, but such structures can be useful for the historical data composition, as well, by forming temporal data layer (oriented on the history). When the database is *in Archive log mode*, database system automatically stores the *REDO* logs in *archive repository* once they are filled. It is done without user intervention and does not impact the statements. The parameter is set in the *MOUNT* mode. The following snippet shows the process in the Oracle RDBMS:

```
SHUTDOWN IMMEDIATE;
STARTUP MOUNT;
ALTER DATABASE ARCHIVELOG;
ARCHIVE LOG LIST;
```

As shown, it is a static parameter, which requires shutting down the database and temporary inaccessibility and connection refusal.

The location of the archive log can be set using *LOG_ARCHIVE_DEST* parameter on the system level. It is recommended to use different disc as the core database:

```
ALTER SYSTEM
SET LOG_ARCHIVE_DEST='LOCATION=C:
\archive_logs' SCOPE=BOTH;
```

The architecture of the system with enabled archive log mode is shown in Fig. 7. Red line shows the log management. Historical data retrieval is depicted by the blue line. First, current valid states are taken (1), followed by the online log processing (2) and archive log operation (3). It is worth mentioning, that if any log data portion is lost, the whole operation fails, even if the particular missing log does not contain the necessary data for the given operation. The process can be automated using *Flashback technology* [31], [32], [33], [34].

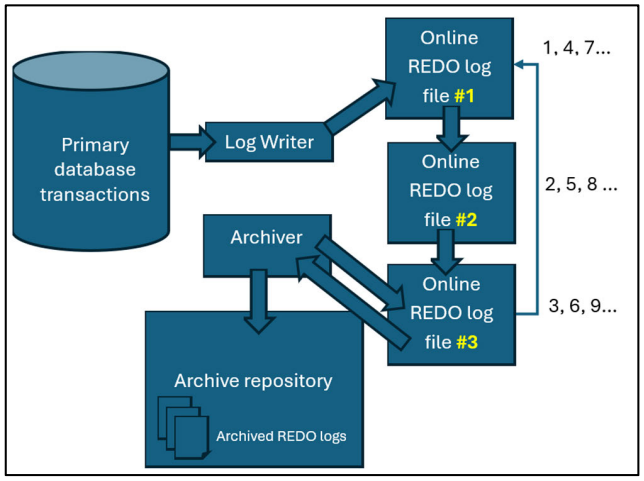



**FIGURE 7. Components for the data archivation – initial online REDO logs and Archive repository, operated by the Log Writer and Archiver background processes.**

### F. FLASHBACK + DATA ARCHIVE

*Oracle Flashback* was first introduced in Oracle Database 9i (released in 2001). Initially, it provided basic functionality such as *Flashback Query* for querying historical data, which allowed users to view data as it was at a specific point in time [31], [35].

In subsequent releases, Oracle enhanced *Flashback technology* with additional features like *Flashback Table* or *Flashback Database*, providing more powerful and flexible recovery options. These features have been gradually improved over time; however, the primary purpose remains always the same – to allow the administrators to recover the database from the user errors. Without using it, a full database restore and recovery would be necessary to apply. Thus, archiving is not primarily intended to provide historical data, although such structures cover all the required data.

Summary of the existing approaches is in the following list [31], [33]:

- *Flashback Query* – gets the query result based on the data as existed in the past. Database content is not impacted.
```
Select *
from TAB
as of timestamp TO_TIMESTAMP(value,
format);
```
- *Flashback Table* - restores a table to its state at the defined point in time.
```
Flashback table TAB
to timestamp TO_TIMESTAMP (value,
format);
```
- *Flashback Database* – restores the whole database to the historical point defined in the *TO TIMESTAMP* clause.
```
Flashback database
to timestamp TO_TIMESTAMP (value,
format);
```
- *Flashback Version Query* – monitors the result set evolution over the defined time frame by using *VERSIONS BETWEEN* clause.
```
Select *
from TAB
versions between
timestamp
TO_TIMESTAMP(valueLEFT, formatLEFT)
and TO_TIMESTAMP
(valueRIGHT, formatRIGHT);
```

### G. SUMMARY OF THE EXISTING APPROACHES

In this section, various approaches and enhancements were discussed. Generally, two techniques were discovered. *Temporal databases* allow to store and monitor temporally framed data based on various precisions and granularities. They build a new ecosystem, in which each row, attribute or synchronization group is enclosed by the valid or transaction time by building uni-temporal or bi-temporal approach. Based on the processed granularity, multiple layers can be present, supervised by the temporal table and module, forming the result set in an appropriate format. In some systems, historical data can be moved into a specific repository. Furthermore, there can be a specific table containing current valid data with no temporal definition. Thanks to that, existing conventional systems can be directly navigated to the conventional database layer. Historical data are then in a separate repository, even in different storage or tablespace. One way or another, it is rather historically oriented database than fully temporal. To make it complex, future valid data need to be operated and stored, as well. It, however, requires automatic transformation of the future plans to the current states at the defined timestamp making it as an extension. Fig. 8 shows the overall architecture of the temporal ecosystem.

The second approach is based on *transaction log management* in an *Archive mode*, in which the historical data can be also obtained. The main advantage is that transaction logs are always present to serve ACID properties. Generally, conventional databases enhanced by the transaction support can be used to build temporal model, if the logs are preserved (e.g. by using enabled Archive log mode. Thus, logs are then available for the security reasons, to be able to provide recovery and to build temporal data). The limitation of that approach is related to the performance by the necessity to scan log files sequentially.

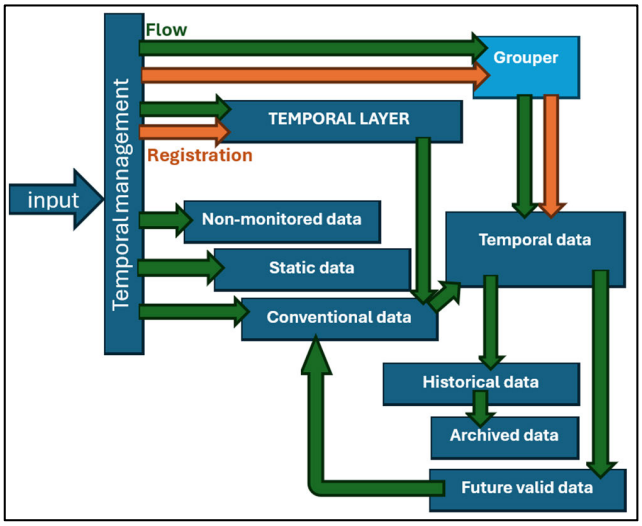



**FIGURE 8.** Temporal ecosystem – all important components and data flow.

Despite the advancements introduced by temporal models in relational databases—such as system-versioned tables and validity time tracking—these approaches still face significant limitations, particularly when compared to the information already present in transaction logs.

Temporal models typically capture only selected aspects of data changes, such as valid time intervals or system-generated timestamps. However, they often omit important contextual metadata that is inherently recorded in transaction logs, including:

- who performed the change (user or system identity),
- what exact operation was executed (INSERT, UPDATE, DELETE),
- the full before-and-after image of modified records,
- transaction boundaries and commit order, and
- exact system timestamps at the level of transaction execution.

This creates a gap in auditability and forensic detail. Temporal tables only reflect state changes, not the operational events that caused them. In contrast, transaction logs—typically used for recovery and replication—contain a much more complete, fine-grained, and trustworthy record of all changes made to the database.

However, because transaction logs are not directly queryable or exposed through standard SQL interfaces, temporal models are often seen as a necessary compromise. They provide a user-accessible history but at the cost of redundancy, limited precision, and the potential for inconsistency if application logic fails to maintain historical correctness.

Furthermore, maintaining temporal tables introduces performance overhead and increases schema complexity, while still falling short of the fidelity and traceability that transaction logs inherently provide. As such, temporal models, while useful, should not be considered a complete substitute for proper change data capture (CDC) mechanisms or log-based auditing systems, especially in contexts requiring strict compliance, traceability, or forensic analysis.

### H. DEFINING CONTRIBUTION FRAME

Based on the state-of-the-art analysis, transaction log management can provide a reliable solution for dealing with the historical data. However, there is a lack of performance caused by the sequential log file scanning necessity. In the next section, our proposed solution is defined to enhance the log management using direct pointers reflecting the object of interest. Thanks to that, sequential scanning necessity is replaced by using direct pointers to the relevant log file blocks.

## IV. PROPOSED SOLUTION

When dealing with the current valid states and states valid in the past, it is not necessary to build a specific temporal system and manage those data on the database layer. Precisely, to make the system reliable, consistent and robust even in case of any error, transaction logs must be preserved for a long time either way. Increasing frequency of reaching Checkpoint does not provide sufficient power, although it brings the data ''image'' at the defined timestamp. However, it strongly increases the demands on the system and overall response time. Moreover, it would significantly increase number of I/O operations, which can form the bottleneck of the system [32].

Historical data can be obtained by scanning the transaction logs, which are available (generally in an unlimited time) in the enabled Archive log mode database system. Sequential scanning would be, however, too demanding. Our proposed solution defines an additional database layer by creating log index (*LogIn*) reflecting the processed objects.

The first solution identifies the referenced object list in the header of the log file (**SOL 1**). The total available size of the online log is slightly lowered by creating the space for the referenced object list. By moving the online log to the archive repository (archiving), original referenced list remain there. The process of the historical data retrieval can be then expressed by the following diagram (Fig. 9). The initial starting state is the currently valid object. Then, the headers of the online logs are scanned to identify blocks of interest covering particular objects. Then, similar activity is done for the archive logs – header of each file is scanned (both scanning activities can be partially processed in parallel).

Scanning is this stream means taking initial blocks of the log files into the memory for the evaluation. To limit the size of the log file header, it took only list of the objects, which are referenced inside, without any additional pointers. Furthermore, each object is present only once in the header. So, to locate the change itself, it is still necessary to sequential scan the whole file. However, it is ensured, that at least one relevant change vector will be always present. To simplify the location process, two enhancements can be implemented:

- Storing number of references for the object, so the sequential scanning can be stopped, if the particular number is reached (**SOL 1a**).
- Storing first position of the referenced object – previous blocks can be skipped (**SOL 1b**).

Fig. 9 shows the data flow diagram.

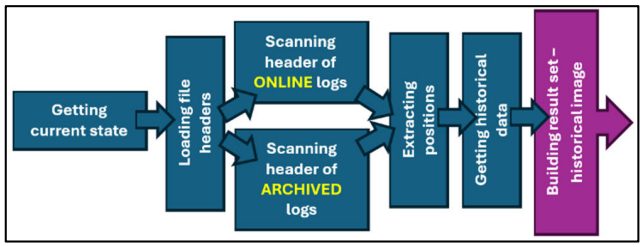



**FIGURE 9.** Essential components necessary to obtain state of the object as existed in the past.

The limitations of the solution:

- Necessity to load file header into the instance memory for each online and archive transaction log.
- Necessity to introduce new memory structure for holding log file headers.
- Uncertain required size for the memory structure - the available size for the log file is dynamic, depending on the number of referenced objects in the log file header.
- Sequential data file scanning necessity in case of the header points to the object of interest.

The second solution (**SOL 2**) limits the sequential log file necessity by holding the address pointers to the positions inside the log file. Although the available space of the log is again reduced, sequential scanning necessity is removed. In this case, the header consists of the *linear list of objects* and references to the positions manipulating those objects. The pointer navigates to the change vector of the appropriate object. Thus, one object can be referenced in the header as many times as the number of changes of the given object covered by the particular file. **SOL 2a** uses a compressed list, so each object is referenced only once, enhanced by the list of addresses sorted based on the *transaction reference* (timeline). **SOL 2b** replaces the flat structure by the B+ tree using object identifier as a navigation key. B+ tree is a balanced index tree, so the efficiency is ensured by the height of the tree. Fig. 10 shows the differences in the file header approach in SOL 2 category.

The address pointer (LogInPo) consists of these elements:

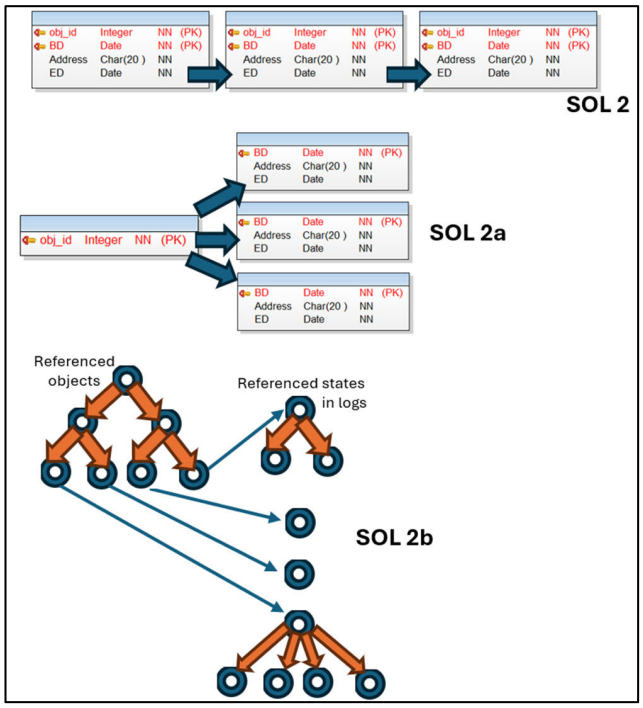



**FIGURE 10.** SOL 2 reference management (file header).

1. identifier of the object,
2. address of the block,
3. position of the change vector inside the block,
4. check sum for the value consistency check purposes (optional, if the LogInPo_checker parameter is enabled):

```
alter system set LogInPo_checker=
{enable | disable [{logical physical}]};
```

Since the above parameter setting impacts the log file structure, it is static. Thus, to apply the change, database system must be shut down properly. Furthermore, before opening the database, even existing archive log files need to be updated. The size of the log can expand, therefore it is important consider the impact of the change properly. The next code snippet shows the step-by-step guide expressing the process of *LogInPo_checker* parameter change. In case of removing check sum, logical or physical approach can be used. By default, *logical approach* is used, meaning, that original check sum values remain in the archive logs, but are not further processed. Thanks to that, the header is not restructuralized and the process of system altering is radically shortened. On the other hand, the log files store irrelevant data (check sums of the already processed rows, which are not later treated in any way). Thus, the efficiency of the log files is considerable, on the other hand, the time during the database system is unavailable (in a *MOUNT* mode), is dropped significantly.

```
1. alter system set LogInPo_checker=
{enable |disable[{logical | physical}]};
2. shutdown;
3. startup mount;
```

4. `apply change`
5. `remap archive logs`
6. `alter database open;`

**SOL 3** expands the above principles by compressing the *LogInPo* addresses. The referenced objects in the log files are not sorted by the timeline (transaction reference) as used in the previous approaches (default option for the transaction log), instead, object clusters are created. Thanks to that, object references are in the consecutive data blocks. Thus, the *LogInPo* addresses can be reduced by pointing to the block reference just once – defining the first and last associated block in a sequence.

Properties of the model SOL 3:

+ log file header size demands are reduced.
+ the data about the object are stored only once.
+ the change vectors for the particular object are physically stored nearby, so the number of blocks to be loaded is reduced, as well.
+ generally, under ideal conditions, one block stores change vectors for one object only.
- there are additional demands for data reallocation.
- additional demands are caused by the sorting.

To summarize the previous proposed solutions, the data in the transaction logs can be sorted based on the transaction reference (timeline) – SOL 2 category or based on the objects as used in SOL 3.

Proposed solution **SOL 4** combines both approaches by defining two interconnected indexes. The first index reflects the temporal position of the change by referring to the timeline, while the second index emphasizes object identifiers as a index key. Both types use B+tree index structure. On the leaf layer, there is a *LogInPo* pointer, but also *TreePo* to the second index. As a consequence, irrespective of the physical structure of the log, direct access can be done either by the object identifier or temporal reference. Flashback query alternative uses static point in time and multiple objects are referenced, by using the second index, while *Flashback VERSION query* benefits from the first index highlighting time references.

### A. LOG FILE HEADER-SUMMARY

The above solutions were based on enhancing transaction log file physically. The header of the file was extended by holding references to the particular positions covering object states. Since the size of the online log is fixed, it is necessary to dynamically increase the file or always to leave sufficient marginal free space. As a consequence, the log file would not be filled completely, which would bring additional storage demands. Moreover, by sequential loading of the file into the memory, number of I/O operations would be increased, as well. Although it could be beneficial from the Select statement point of view (querying), additional demands would be present, when archiving and during the referencing – creating file header. Fig. 11 shows the flow of the log file header management.

In this context, it is worth mentioning the requirement for the prohibited history and purge operations, which require

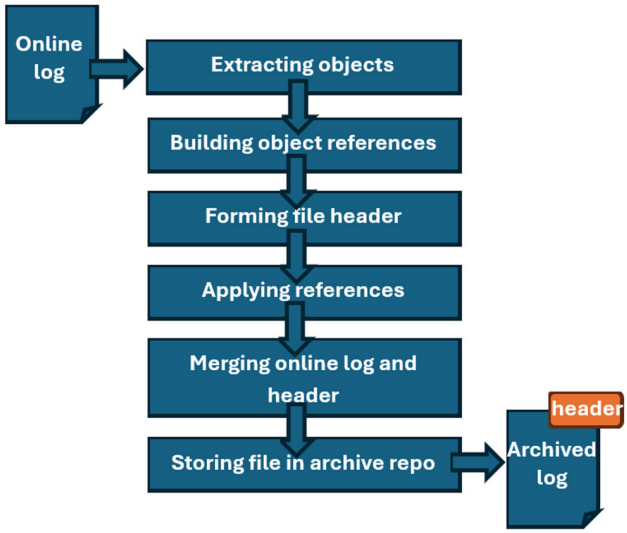



**FIGURE 11.** Step-by-step guide, how the header of the log is composed and filled.

object state to be stored for the precisely defined time period and after the time period is elapsed, particular object reference must be deleted. For the already stated solutions, it would require rebuilding file header, either by vacuuming the content – freeing the values without rebuilding the header itself or by compressing freed space to lower the storage demands. One way or another, provided solutions are not optimal. Mostly in the dynamic systems, in which many object states and present, provided solution would not be suitable and enough robust, due to the rate of the header implementation efficiency. Furthermore, it can even happen, that the input queue for the archiving can be full causing delays in the processing. In the event of a large increase in the number of changes (new states), this system may even completely collapse. Namely, online log cannot be overwritten before it is archived. However, archiving itself is preceded by the log header composition, which can be demanding. One of the possible solutions is to extend the number of online logs in the instance memory, which, however requires additional memory storage demands, as well as the costs for the formatting and operations.

Database mapping layer uses a different approach by extracting log references directly in the database.

### B. DATABASE MAPPING LAYER

Database mapping layer (**SOL 5**) is an introduced solution to address previous limitations. Instead of forming additional structure part of the log file header, particular references are directly associated with the database structure. Precisely, archive repository is outside of the database system itself (no part of the database for the security and recoverability reasons), however, the structure itself is protected by the database. The files are supervised by the database owner, so the data manipulation must be done from the database

layer, not outside (file system). Thanks to that, archiving is straightforward – just a 1:1 copy from the online log repository to the archive. There are no additional demands, nor operations done during the archiving, so there is no risk of online logs being overloaded. The whole management and efficiency is ensured and protected by the database layer. The database itself stores list of objects, which must be temporally monitored. Individual states of the objects are referenced based on the validity timeline. Change vector reference is then stored as a single unit forming the positional address. It refers to the file identifier, block inside that, temporal borders and positions of the change vector, separately for the original state (*UNDO*) and new version (*REDO*). Additionally, direct link to the consecutive state is stored in form of the address reference.

Database structure is delimited by the flat table, supervised by three index strategies:

- indexing objects,
  indexing states for each object
- indexing states based on the temporal timeline reference.

Fig. 12 shows the overall architecture and basic data flow. In that case, four temporal frames can be identified and handled:

- timestamp of transaction start (*Tbd*),
- timestamp of inserting new state (*Tins*),
- timestamp of transaction approval (*Tapr*),
- validity of the state (*BD, ED*).

All these values are stored in the multilayer indexing structure. The archiving itself is just a copy to the specific archive repository and does not need to be synchronized with external routines. Thanks to that, online logs are almost immediately available after filling up, reducing the demands and costs for handling online logs (no need to extend the number of log file groups). Indexing of the archive log is done autonomously in a separate stream, ending by notifying temporal system, that content is generally available and indexed. In the meantime, particular archive log would be necessary to be scanned sequentially, however, it would be done just once. As soon as the file is scanned sequentially, it is automatically indexed.

Note, that the proposed approach can be directly applied not only to archive repository, but also the online logs. In terms of archiving, just the location prefix would be changed (Fig. 13).

The data loading process is depicted in Fig. 14, reflecting the timeline. Data processing automatically invokes logging data into Online log. Transaction approval launches process of archivation. Once archived, original online log is available for rewriting, even the log indexing has not finished, yet. Thanks to that, throughput of the whole system is risen. Besides, memory requirements are lowered.

Fig. 15 declares the process of data retrieval and elements impacting the result set composition. Blue arrows represent data flow from the content creation of view, while red arrows express data flow from elements building temporal result set.

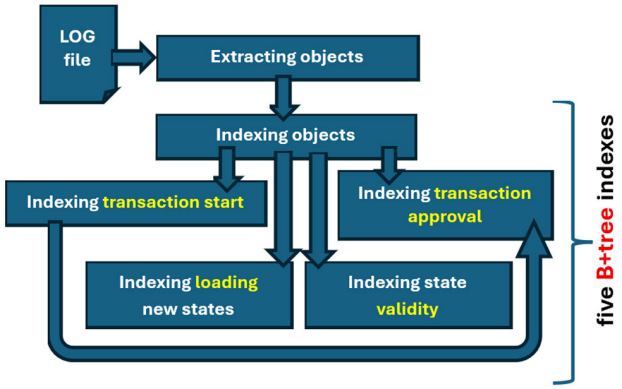



**FIGURE 12.** Log file indexing enhanced by the transaction data – indexing transaction start, loading, approval and validity reflection.

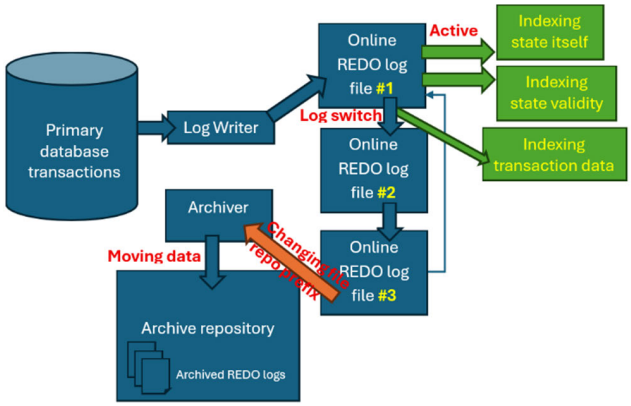



**FIGURE 13.** Online indexing – delimited by changing the repo prefix associated with the data moving.

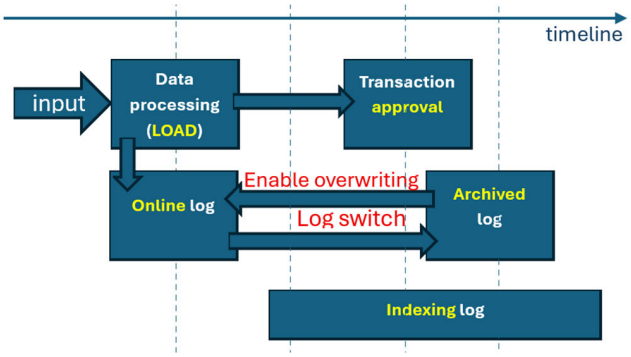



**FIGURE 14.** Data loading – timeline reference of the online REDO log creation, archiving and indexing.

## V. PERFORMANCE

The performance evaluation study was conducted in a controlled testing environment featuring a server configured with the following specifications:

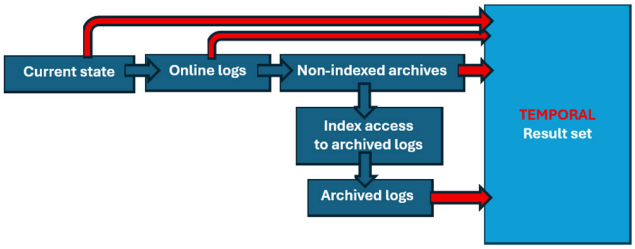



**FIGURE 15.** Data retrieval – components building temporal result set.

- *Central processing unit*: AMD Ryzen 5 PRO 5650U with Radeon Graphics, 2.30 GHz
- *Memory*: 64 GB DDR4 (2 × 32 GB, 3200 MHz, CL20)
- *Storage*: 2 TB NVMe SSD with a read/write speed of 3500 MB/s
- *Operating System*: Windows 11 Pro, version 24H2
- *Database System*: Oracle Database 23ai Free, Release 23.0.0.0.0 – Production Version 23.4.0.24.05

For the computational study, a real-world dataset from the aviation domain was utilized. The dataset comprises three key components [36], [37], [38]:

- *Planned flight routes*, outlining the intended paths of aircraft.
- *Actual flight routes*, incorporating positional data of aircraft and other objects, as well as the operational status of various flight regions.
- *Flight monitoring information*, detailing the assignment of aircraft to Flight Information Regions (FIRs), based on recorded entry and exit times.

A *Flight Information Region* (FIR) [38], [39], [40] is a designated area of airspace where a specific country's *Air Traffic Control* (ATC) authority is responsible for providing Flight Information Service (FIS) and alerting service. Defined by the *International Civil Aviation Organization* (ICAO), FIRs encompass the entire globe, including both land and oceanic areas [41], [42]. The primary functions of FIRs include [43], [44]:

- ensuring the safety and efficiency of air traffic by delivering critical information and coordinating emergency responses,
- managing and controlling airspace,
- supporting route planning and optimization to reduce environmental impact.

The dataset used in this study comprised 10,000 flights. On average, each flight included approximately 1,000 records for both the planned and actual routes.

The evaluation study can be divided into two experiments pointing to the performance and costs. In the sensor-based environment, there are two key operations related to the database layer, process management and optimization – data loading and consecutive data retrieval [45], [46]. Experiment 1 deals with the data loading by pointing to the five core solutions, which have already been described. All the data were conventionally treated, the temporal aspect was covered by the transaction time flag, emulating uni-temporal solution. The data retrieval process (experiment 2) is divided into three groups:

- getting current valid state, which does not require log to be handled and operated,
- getting historical data image at defined timepoint, which starts with current state and applies historical changes by reverting object to the state valid in the past,
- monitoring evolution during the defined time frame.

All the results and solutions were compared and referenced to the uni-temporal model covering validity. For the clarity, results are shown in percentage.

Tab. 2 shows the results for the data loading. It is evident, that indexing brings a significant improvement compared to the solution managing references in the file header. Regardless of the physical architecture implemented for the header, more than 45% improvement can be identified. First, let´s consider the processing time. Any extension of the core solution brings additional demands, primarily caused by the physical header, which is more sophisticated, either sorted in the linked-list or by using searching tree. Based on the results, solutions 1 and 2 require approximately 50% additional costs. It is done by file header reconstruction, which is also dynamic in terms of size. Solution 3 is the worst, because it reconsiders the data in the files by moving the position of the change vectors. For each new tuple, the whole file needs to be updated. The header points to the first block dealing with the particular object, individual changes are clustered, so the consecutive blocks are loaded. Although the loading can be done in parallel, as evident, the loading is the most demanding. Solution 5 requires additionally only a little more than 3%. The whole structure for the log position references is located in the database, so all optimization techniques and database indexes can be directly used and implemented. Furthermore, the data can be pre-fetched and stored in the instance memory. The archivation process is separated from the main transaction, so no additional waiting time is necessary. To conclude that, the data mapping layer (SOL 5) provides the best solution in terms of the processing time of the data loading operation, which is not, however, true, if storage demands are considered. Namely, although it provides one of the better solutions, it is not the best, since it requires data table, but also indexing of the state references in the time line. Thus, SOL 5 requires additional 8.83%. By replacing the (heap) table and index by the index-organized table, storage demands can be lowered to 105.63% in total, which is almost the same compared to the SOL 4. Storage demands for the file header management range from additional 5% up to more than 15% for the SOL1b, which is a bit specific, because it handles all the references and position of the first block separately. The total costs is a metric emphasizing storage demands, processing time, as well as all used resources for the processing. In terms of the performance, proposed mapping layer provides the best solution, which requires less than 10% additional resources. The worst solution is provided

**TABLE 2.** Data loading results.

| Data loading | Storage demands (%) | Processing time (%) | Costs (%) |
|---|---|---|---|
| Temporal solution | 100 | 100 | 100 |
| SOL 1 | 112.63 | 160.06 | 139.85 |
| SOL 1a | 114.04 | 164.89 | 142.32 |
| SOL 1b | 115.61 | 147.66 | 137.29 |
| SOL 2 | 108.01 | 149.96 | 132.12 |
| SOL 2a | 106.84 | 151.43 | 126.53 |
| SOL 2b | 105.12 | 152.05 | 130.71 |
| SOL 3 | 103.19 | 205.23 | 173.71 |
| SOL 4 | 105.72 | 116.72 | 110.23 |
| SOL 5 | 108.83 | 103.03 | 109.65 |

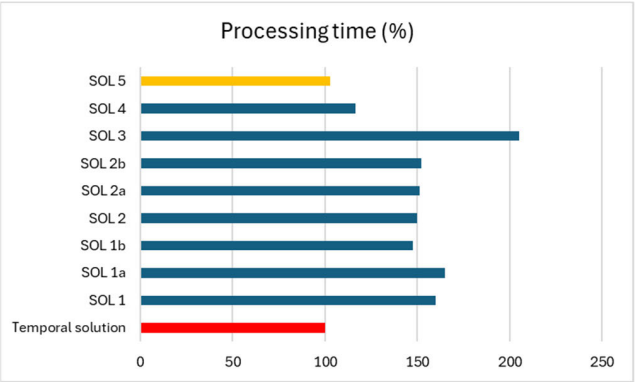



**FIGURE 17.** Data loading – processing time.

by the SOL 3 using clustering, which remaps the positions dynamically. The log header itself cannot be implemented, the references cannot be stored, because they strongly evolve. Instead, only the first position of the reference of the object is stored. It is worth mentioning, that even there can be free space in the cluster – blocks do not need to be totally filled, from the storage demands perspective, it is not relevant and additional demands are only 3.19%. Tab. 2 shows the results. Storage demands are depicted in chart format in Fig. 16, while processing time requirements are reflected in Fig. 17.

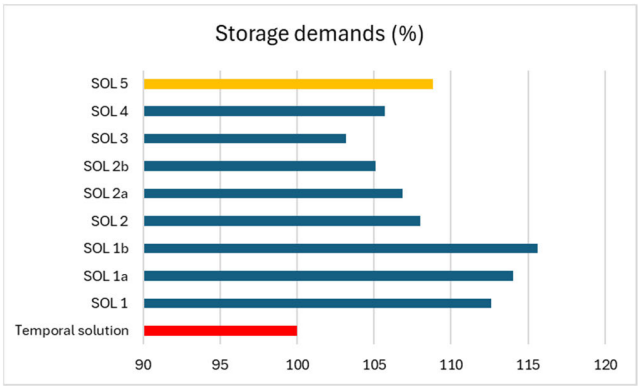



**FIGURE 16.** Data loading – storage demands.

Fig. 17 shows the time costs. From the perspective of individual extensions of the base core solutions SOL 1 and SOL2, the time component is not significantly changed. On the other hand, if we look at the disk storage area, when using extensions storage requirements are always reduced and thus the total hardware costs are reduced, as well.

The second evaluation stream emphasizes data retrieval process. The reference model is an attribute oriented temporal approach, in which each change is recorded in the temporal layer separately. This is the most suitable solution for the aviation data, because the data sent from sensors are not time synchronized. So, generally, to get the object composition, individual attributes need to be extracted and merged. However, current valid states are directly accessible in the conventional layer. As evident from the results, getting current state is not impacted by any temporal architecture and the results are almost the same. The only difference is in the last solution, which models the mapping layer. In that case, even better solution is obtained, compared to the ordinary temporal solution. The reason is, that the whole structure is reduced, database stores only current valid data and temporal references. The key benefit is associated with the complex indexing, so the data are directly accessible in the lime line reference. Proposed mapping layer (SOL 5) lowers the processing time demands for getting current state by eight percent. This is because of the precise index usage. All other solutions have the difference lower than 1%, which is not worth further investigation.

When dealing with the historical data, it is necessary to scan log data to create an image valid in the past. The easiest way is to build static image, which is defined by the historical timestamp. Sequential log scanning is absolutely inappropriate solution. Primarily, it is not scalable, at all.

The amount of data is constantly growing, objects evolve over time, and therefore the amount of historical data can very soon exceed a "milestone", so obtaining temporal data from logs would not be possible in a reasonable time. Of course, there would also be enormous costs associated with the processing itself, sequential scanning and loading blocks into memory. In this context, the most important operation is the I/O work between the database as a physical storage and the memory instance in which the evaluation takes place. The key benefit of the log management is just the precision, only relevant data are stored in the context of the change vector – non-changed values are not preserved, nor referenced in the log files, so there are no duplicate tuples located and stored. The key for the optimization lies in the ability to identify and locate blocks of interest consisting of the change vectors for the particular objects directly without the necessity to scan the file block by block in a sequential manner. There are various techniques and enhancements in terms of file header composition. The results are in Tab. 3.

Data retrieval process in the temporal environment points to the data image monitoring by tracking the changes and

**TABLE 3.** Data loading results.

| Data loading | Getting current state | Getting historical image | Monitoring changes |
|---|---|---|---|
| **Temporal solution** | **100** | **100** | **100** |
| **SOL 1** | 100.68 | 74.03 | 92.04 |
| **SOL 1a** | 100.35 | 72.42 | 83.33 |
| **SOL 1b** | 100.51 | 73.56 | 84.18 |
| **SOL 2** | 100.59 | 69.09 | 74.03 |
| **SOL 2a** | 100.61 | 64.31 | 69.92 |
| **SOL 2b** | 100.31 | 59.74 | 65.01 |
| **SOL 3** | 100.94 | 52.98 | 58.71 |
| **SOL 4** | 99.98 | 44.32 | 49.57 |
| **SOL 5** | 92.03 | 39.81 | 44.21 |

impacts to the object state itself. Thus, it is not just about the change monitoring, but it is inevitable to reflect consequences by introducing importance and significance. The overall goal is to track the object states over time, to manage and register changes and data image composition. Besides, it is important to build consistent data image at the defined timestamp. To reflect the performance, processing time demands were evaluated for the static timestamp data image composition. The attribute level temporal system is a reference model getting 100%. SOL 1 does not bring significant benefit, compared to other proposed techniques. Namely, for each log file, all blocks associated with the header must always been loaded, even if the data log does not contain relevant data for getting the image, but that is not known in advance – without the necessity to scan the header. SOL 1a brings only a slight benefit by shortening the scanning, because the number of references for the object is stored directly in the header. Thanks to that, system can choose the best suitable way to identify and track changes. Even though there are references in the header file, it is possible that the log management system still opts for sequential scanning. In dynamic systems where the frequency of changes is not clearly defined, this can bring significant benefits. SOL 1b takes almost the same results as SOL 1a and pure SOL 1 solution, respectively. Header optimization defined in SOL 2 and its enhancements brings a relevant benefit, since the header is compressed and requires less number of blocks to be loaded and evaluated. It is worth mentioning, that the header is also block oriented and its scanning must be done in the instance memory, so the loading is inevitable. Pure SOL 2 lowers the demands to 69.09%, compared to the attribute-oriented approach. This is a result of two basic activities – reducing object state composition from individual attributes and identification of the real attribute change, because the change vector stores only relevant data, optionally enhanced by the Epsilon module tracking only significant attribute value changes. By compressing the header, the time cost can be reduced even more significantly. Despite the fact that data decompression is necessary in the result set composition process, efficiency is ensured by reducing the number of blocks to load and I/O operations that have the greatest impact on performance. The compress list introduced in SOL 2a reduces the processing time costs by 6.91%. B+ tree data structure formatting the log file header is even better, reducing the processing time demands by 13.53%. The key is the object itself, extended by the state reference to the timeline.

Clustering is the process of grouping similar data points together based on the object references. Generally, individual changes for the object can be distributed in the log file randomly, so many blocks can be necessary to be loaded. The loading and mapping is block oriented, so even there is just one piece of data taking a few bytes, the whole block must be loaded. By using clustering, blocks are primarily filled by the changes associated with the particular object. So the number of blocks to be loaded is strongly reduced. On the other hand, you may see significantly increased costs for acquiring and building a cluster itself. Namely, data retrieval processing time costs are 52.98%, which reflects more than 47% improvement. Processing time demands for the cluster composition are increased by more than 100%. The best solution was obtained by moving log reference management to the database layer, instead of the log file header defined by the SOL 4 and SOL 5. Precisely, SOL 5 lowers the processing time demands by 60.19% compared to the pure temporal solution. The third column of the Tab. 3 shows the results. Graphical representation of the results is in Fig. 18.

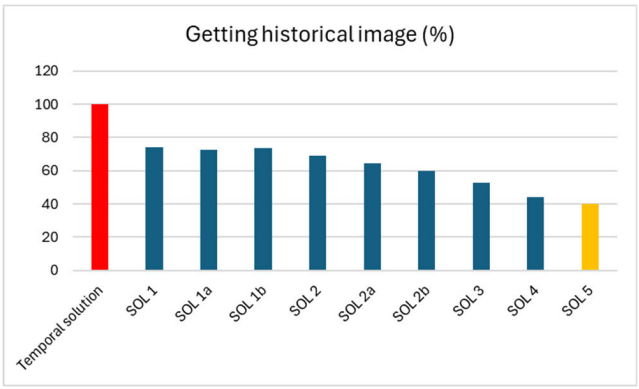



**FIGURE 18.** Data retrieval – getting historical image.

When dealing with the temporal change monitoring in the defined temporal range, the best results were reached in mapping layer (SOL 5) applied to the database layer, taking improvement by 55.79%. SOL 4 provides also valuable results by implementing two core indexes, for the object and individual states. The results are relevant and consistent, even though the data in the indexes are not sorted based on the time line, just by the object and states themselves, so the temporal references must be obtained separately. SOL 1 category requires sequential log scanning. Its enhancements limit the number of loaded blocks by getting 9.46% and 8.54% improvement respectively. Considering clustering, it brings a relevant contribution and performance reducing the processing time costs by 41.29%. Precisely, it is ensured, that data

tuples are in the consecutive log blocks, so any other blocks can be omitted from the evaluation. However, as stated, clustering is fairly poor for building and maintenance. So the most suitable solution again seems to be SOL 5, based on the database mapping layer. It uses database indexes, which also ensure scalability. The total demands of the SOL 5 are 44.21%. Fig. 19 shows the results in form of the chart.

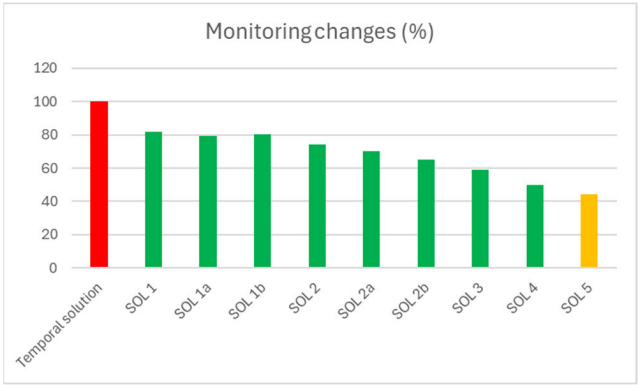



**FIGURE 19.** Data retrieval – change monitoring – temporal range.

Historical data reflection is compared in Fig. 20. It depicts static data image valid in the past and historical data monitoring. Individual operations provides almost the same results by respecting the difference ratio.


**FIGURE 20.** Data retrieval – historical data processing.

### A. SUMMARY

Efficient data loading and retrieval processes are critical components of modern database systems, serving as the technological foundation for applications and overall data management. Today, it is no longer sufficient to manage only the current valid state of data; instead, the full evolution of data objects over time must be captured and maintained [47], [48]. This comprehensive historical tracking enables informed decision-making, accurate evaluations, and reliable future predictions. Data-driven decisions contribute to sustainability, durability, consistency, and verifiability within data-centric systems.

Temporal databases address these needs by providing mechanisms to store, manage, and evaluate the states of data objects within defined validity timeframes. However, implementing temporal features typically requires rethinking existing architectures and often demands significant modifications to current solutions. Specifically, conventional database architectures must be extended to support temporal functionalities, including new data processes, memory structures, and most notably, a modified data interface. As a result, existing applications are frequently incompatible without direct code modifications.

Traditional database systems are inherently tied to transactional support, typically managing change through log files composed of change vectors. This paper focuses on long-term optimization strategies for log file management, proposing a solution that retains compatibility with the existing database layer while supporting historical data management through transaction logs in an auditable environment. Rather than sequentially processing log files—a method that is resource-intensive, inefficient, and inherently unscalable—this work explores several optimization techniques aimed at improving log data access.

**Solution 1 (SOL 1)** introduces a referenced object list stored in the header of the log file. While this allows for quick determination of whether a log file contains a particular object, it does not store positional information. If the object is listed in the header, the system must still perform a sequential scan to locate the relevant change vectors. Enhancements such as storing the count or positions of first and last references can slightly reduce overhead, but large temporal ranges still require loading and scanning many blocks, maintaining the need for sequential access.

**Solution 2 (SOL 2)** utilizes structured address lists in the form of linear linked lists, compressed lists, or B+ trees. This method eliminates the need for sequential scanning by directly referencing change vector positions. However, this approach can lead to excessively large log file headers, potentially limiting the overall log content due to static file size constraints.

**Solution 3 (SOL 3)** introduces clustering, wherein change vectors for a specific object are stored in consecutive data blocks. Only the position of the first block is recorded. During data retrieval, the system reads from the initial block and continues until it encounters a block without the object reference, at which point processing stops. While this improves retrieval performance, it significantly slows down data loading operations due to the need to maintain block adjacency.

The primary contribution of this paper is **Solution 5 (SOL 5)**, which introduces a **database mapping layer**. This layer manages data references directly within the database, eliminating the need to sequentially scan log files during retrieval. Importantly, the log file header remains unchanged, ensuring no impact on file size. Performance is achieved through a set of indexes that reflect both object identities and temporal dimensions. These indexes are maintained separately and updated during the archiving process in an independent

transaction stream, allowing the online log to be rewritten immediately after archiving, with no delay.

Scalability is ensured through the index layer itself, which can leverage traditional database optimization strategies such as partitioning and data distribution. The additional storage overhead introduced by the mapping layer is minimal—only **8.83%**, which is offset by the unchanged structure of the log file headers. In terms of processing time, the proposed solution incurs just **3.03% additional cost** during data loading.

However, the major benefit of SOL 5 lies in its impact on historical data retrieval. The presence of temporal indexes reduces processing time by **60.19%** for static historical data images and by **55.79%** for managing data evolution. This is enabled by fine-grained tracking of change vectors at the attribute level, removing the need for reconstructing object states from individual attributes—a common limitation in attribute-oriented temporal models.

## VI. CONCLUSION

Temporal databases are designed to manage data evolving time-based states of the objects. They store not just the data values, but also time period references, during which those values are valid or have been recorded. They allow to store and retrieve past states of the data, enabling analysis, forecasting and data mining over time. Currently, many systems require temporal audit trails and change monitoring in a proper way by focusing on the versioning and data corrections. Instead of overwriting old data, you can add new valid time intervals, maintaining a full record of all versions. Temporal databases enforce consistency of time intervals, preventing overlapping or conflicting entries for the same entity.

There is a huge correlation between the temporal modeling and transaction support in the conventional paradigm. A conventional transaction is a unit of work that must be atomic, consistent, isolated, and durable. A transaction doesn't just modify data—it records the time context of each change. Old data tuples are preserved in the transaction logs to manage consistency and concurrency control.

A transaction log is a critical component of a database system that records all changes made to the data. It ensures data integrity, recovery, and auditing, especially in the event of system failures. The structure can be divided into UNDO and REDO elements, forming the change vector stored primarily in the Online log repository, optionally preserved for a long time in Archive repository, supervised by the Archiver background process. Although logs are primarily used for transaction management, they hold important data about the changes and state evolution. Therefore, in this paper, a robust novel solution managing transaction logs efficiently is introduced. Various structures and enhancements are introduced, presented and discussed aiming to find the most suitable solution in terms of the data complexity and scalability. Initially, solution extending log file header was discussed, limiting the necessity to scan the whole log file set sequentially. Although it brings significant benefits by using direct locations – addresses to the blocks and positions referencing the object state change vector, the size of the whole structure is unpredictable. It consequences in inability to appropriately set the size of the memory structure serving as a buffer for processing logs. Moreover, various security, time-limited data management options and purge operations can cause necessity to rebuild the structure, to remove references and compress the size, which can be too demanding for the operational level.

The proposed data mapping layer holds all the references to the logged data directly in the database, optimized by using five indexes – object definition and four temporal B+trees, defining the validity, timestamp of loading and transaction borders (transaction start, transaction approval). The whole structure is block-oriented and optimized for the dynamic access and changes. It also effectively implements aspects of prohibited history.

Based on the proposed performance evaluation study, proposed solution brings significant performance benefits, compared to the existing log management approach forcing system to scan logs sequentially. Furthermore, the total processing costs are at the level of temporal systems, however, no temporal layer and migration between conventional and historical data is necessary. No precision level modeling definition is necessary, since the change logs refer just to the actually changed values.

The limitations of the proposed solution dealing with the data mapping cover the following aspects:

- *Effectiveness of log content* – log files contain also other descriptive and hose-keeping data about the transactions and instance monitoring itself. Although they are not indexed, nor loaded and scanned, from the long-term perspective, they can require a significant component of disk storage requirements.
- *Data recoverability*– if there is a requirement to drop historical reference to the object state or the whole object, generally, it is implemented on the index level, but the log content still remains original, since it would require recalculating positions and references. So, as a consequence, even deleted object states can be still recovered by sequential scanning of the log. It can impact the reliability of the system, but also impacts the storage demands.
- *Different structure*– transaction logs apply different structures compared to table data, so the database optimization techniques cannot be directly applied.
- Unavailability to change block size of the log.
- *Bottleneck of the system*– in the dynamic systems with varying ratio of the change frequency, inappropriate log settings can create a bottleneck in the system. Namely, without a free online log, transactions cannot be processed and the entire system hangs.

In the future research, we will emphasize the aspect of reliability by allowing to compress log files just to data of interest. The focus will also be done of the prohibited history to imple-

ment physical delete from the archived log files. The impact of the index key definition will be investigated, as well. It is assumed, that implementing access rate as a priority of the index can improve performance of the access to the log files.

## REFERENCES


[1] W. Penzo, "Rewriting rules to permeate complex similarity and fuzzy queries within a relational database system," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 2, pp. 255–270, Feb. 2005, doi: 10.1109/TKDE.2005.33.

[2] J. J. Koh, "Relational database schema integration by overlay and redundancy elimination methods," in *Proc. Int. Forum Strategic Technol.*, Oct. 2007, pp. 610–614, doi: 10.1109/IFOST.2007.4798673.

[3] L.-L. Wei and W. Zhang, "A method for rough relational database transformed into relational database," in *Proc. IITA Int. Conf. Services Sci., Manage. Eng.*, Zhangjiajie, China, Jul. 2009, pp. 50–52, doi: 10.1109/SSME.2009.79.

[4] J. Zhao and M. Li, "Large-scale concurrency for MySQL database performance analysis," in *Proc. 8th Int. Conf. Adv. Algorithms Control Eng. (ICAACE)*, Shanghai, China, Mar. 2025, pp. 2098–2102, doi: 10.1109/ICAACE65325.2025.11020302.

[5] V. Subramanian, "Data extraction," in *Applied Machine Learning for Data Science Practitioners*. Hoboken, NJ, USA: Wiley, 2025, pp. 39–55, doi: 10.1002/9781394155408.ch3.

[6] S. Yuan, L. Zhao, J. Qi, and Y. Zhang, "Architecture design based on big data storage technology for launch vehicle test data," in *Proc. 4th Int. Symp. Comput. Appl. Inf. Technol. (ISCAIT)*, China, Mar. 2025, pp. 2146–2150, doi: 10.1109/ISCAIT64916.2025.11010775.

[7] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970, doi: 10.1145/362384.362685.

[8] Y. Luo, Y. Zhang, Z. Liu, G. Lai, Y. Wen, and W. Gao, "An efficient data synchronization tool design and implementation based on data flow," in *Proc. 7th Int. Conf. Softw. Eng. Comput. Sci. (CSECS)*, Taicang, China, Mar. 2025, pp. 1–5, doi: 10.1109/CSECS64665.2025.11010010.

[9] L. Peng and Z. Chen, "A high-performance scientific database supporting in-situ data query and accessing," in *Proc. 4th Asia Conf. Algorithms, Comput. Mach. Learn. (CACML)*, Guangzhou, China, Mar. 2025, pp. 1–6, doi: 10.1109/CACML64929.2025.11010937.

[10] G. Fu, H. Zhu, Y. Feng, Y. Zhu, J. Shi, M. Chen, and X. Wang, "Fine grained transaction log for data recovery in database systems," in *Proc. 3rd Asia–Pacific Trusted Infrastructure Technol. Conf.*, Wuhan, China, Oct. 2008, pp. 123–131, doi: 10.1109/APTC.2008.7.

[11] T. Hara, K. Harumoto, M. Tsukamoto, and S. Nishio, "Database migration: A new architecture for transaction processing in broadband networks," *IEEE Trans. Knowl. Data Eng.*, vol. 10, no. 5, pp. 839–854, Sep. 1998, doi: 10.1109/69.729745.

[12] Z. Wei and J. Zhang, "A long-lived transaction model and mechanism in grid database," in *Proc. 4th Int. Conf. Comput. Sci. Educ.*, Jul. 2009, pp. 788–791, doi: 10.1109/ICCSE.2009.5228156.

[13] V. Leis, A. Kemper, and T. Neumann, "Scaling HTM-supported database transactions to many cores," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 2, pp. 297–310, Feb. 2016, doi: 10.1109/TKDE.2015.2411272.

[14] L. V. Orman, "Transaction repair for integrity enforcement," *IEEE Trans. Knowl. Data Eng.*, vol. 13, no. 6, pp. 996–1009, Nov. 2001, doi: 10.1109/69.971192.

[15] P. Kovačovič, R. Pirník, P. Peniak, L. Hrmo, J. Krško, and M. Míkvy, "Transformation of locally implemented data warehouses to the cloud," in *Proc. 26th Int. Carpathian Control Conf. (ICCC)*, May 2025, pp. 1–6, doi: 10.1109/ICCC65605.2025.11022843.

[16] D. McCreary and A. Kelly, *Making Sense of NoSQL: A Guide for Managers and the Rest of Us, Manning*, 2013. [Online]. Available: https://www.manning.com/books/making-sense-of-nosql

[17] O. Abahussain and A. Alqaddoumi, "DBMS, NoSQL and securing data: The relationship and the recommendation," in *Proc. Int. Conf. Innov. Intell. Informat., Comput. Technol. (3ICT)*, Sakheer, Bahrain, Dec. 2020, pp. 1–6, doi: 10.1109/3ICT51146.2020.9311958.

[18] B. Boutsinas, "On defining OLAP formulations," *IMA J. Manage. Math.*, vol. 16, no. 4, pp. 339–354, Oct. 2005, doi: 10.1093/imaman/dpi012.

[19] C. Blanco, E. Fernández-Medina, and J. Trujillo, "Modernizing secure OLAP applications with a model-driven approach," *Comput. J.*, vol. 58, no. 10, pp. 2351–2367, Oct. 2015, doi: 10.1093/comjnl/bxu070.

[20] X. Zhao and Z. Huang, "A quality evaluation approach for OLAP metadata of multidimensional OLAP data," in *Proc. 2nd IEEE Int. Conf. Inf. Manage. Eng.*, Chengdu, China, Apr. 2010, pp. 357–361, doi: 10.1109/ICIME.2010.5477583.

[21] R. Gholivand, P. Goudarzi, and D. Maleki, "An improved hybrid data warehousing architecture for cloud service providers," in *Proc. 29th Int. Comput. Conf., Comput. Soc. Iran (CSICC)*, Feb. 2025, pp. 1–5, doi: 10.1109/CSICC65765.2025.10967465.

[22] A. V. Nguyen and S. Mavromoustakos, "Improving the performance of data warehousing with column databases," in *Proc. Int. Conf. Electr., Comput. Energy Technol. (ICECET*, Jul. 2024, pp. 1–5, doi: 10.1109/ICECET61485.2024.10698745.

[23] P. Thamjaroenporn and T. Achalakul, "Big data analytics framework for digital government," in *Proc. 1st Int. Conf. Big Data Anal. Practices (IBDAP)*, Bangkok, Thailand, Sep. 2020, pp. 1–6, doi: 10.1109/IBDAP50342.2020.9245461.

[24] P. Gonzalez-Alonso, R. Vilar, and F. Lupiañez-Villanueva, "Meeting technology and methodology into health big data analytics scenarios," in *Proc. IEEE 30th Int. Symp. Comput.-Based Med. Syst. (CBMS)*, Thessaloniki, Greece, Jun. 2017, pp. 284–285, doi: 10.1109/CBMS.2017.71.

[25] A. Londhe and P. P. Rao, "Platforms for big data analytics: Trend towards hybrid era," in *Proc. Int. Conf. Energy, Commun., Data Anal. Soft Comput. (ICECDS)*, Chennai, India, Aug. 2017, pp. 3235–3238, doi: 10.1109/ICECDS.2017.8390056.

[26] N. W. Grady, J. A. Payne, and H. Parker, "Agile big data analytics: AnalyticsOps for data science," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2017, pp. 2331–2339, doi: 10.1109/BIGDATA.2017.8258187.

[27] T. K. Balaji, V. S. Sundaram, H. G. Balaji, and M. V. Kavitha, "Eazy trip: Enhancing travel convenience through intelligent route planning and real-time data," in *Proc. Int. Conf. Comput. Commun. Technol. (ICCCT)*, Chennai, India, Apr. 2025, pp. 1–8, doi: 10.1109/ICCCT63501.2025.11019624.

[28] H.-C. Wei and R. Elmasri, "Study and comparison of schema versioning and database conversion techniques for bi-temporal databases," in *Proc. 6th Int. Workshop Temporal Represent. Reasoning*, Orlando, FL, USA, 1999, pp. 88–98, doi: 10.1109/TIME.1999.777976.

[29] C. Njovu and M. T. Ibrahim, "Taxonomy of bi-temporal events data semantics," in *Proc. 15th Int. Workshop Database Expert Syst. Appl.*, 2004, pp. 592–595, doi: 10.1109/DEXA.2004.1333539.

[30] M. Kaufmann, P. M. Fischer, N. May, C. Ge, A. K. Goel, and D. Kossmann, "Bi-temporal timeline index: A data structure for processing queries on bi-temporal data," in *Proc. IEEE 31st Int. Conf. Data Eng.*, Apr. 2015, pp. 471–482, doi: 10.1109/ICDE.2015.7113307.

[31] M. Kvet, *Developing Robust Date and Time Oriented Applications in Oracle Cloud: A Comprehensive Guide To Efficient Date and Time Management in Oracle Cloud*. Birmingham, U.K.: Packt Publishing, 2023.

[32] J. Li, X. Li, G. Liu, and Z. He, "Log management approach in three-dimensional spatial data management system," in *Proc. 18th Int. Conf. Geoinformat.*, Beijing, China, Jun. 2010, pp. 1–5, doi: 10.1109/GEOINFORMATICS.2010.5568028.

[33] C. Teixeira, J. B. de Vasconcelos, and G. Pestana, "A knowledge management system for analysis of organisational log files," in *Proc. 13th Iberian Conf. Inf. Syst. Technol. (CISTI)*, Caceres, Spain, Jun. 2018, pp. 1–4, doi: 10.23919/CISTI.2018.8399229.

[34] M. Singh and N. Kaur, "Log management based on three dimensional spatial database and log management techniques," in *Proc. Int. Conf. Comput., Electron. Electr. Technol. (ICCEET)*, Nagercoil, India, Mar. 2012, pp. 837–840, doi: 10.1109/ICCEET.2012.6203918.

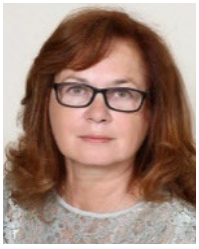
[35] S. He, G. Liu, Z. He, and Z. Weng, "Design and implementation of log management module in three-dimensional spatial database management system," in *Proc. 18th Int. Conf. Geoinformat.*, Beijing, China, Jun. 2010, pp. 1–5, doi: 10.1109/GEOINFORMATICS.2010.5567648.

[36] W. Yan, M. Han, Z. Tian, W. Zheng, and X. Gao, "Research on intelligent monitoring and management technology integrating airport security check business with GIS platform," in *Proc. Int. Conf. Digit. Anal. Process., Intell. Comput. (DAPIC)*, Feb. 2025, pp. 328–332, doi: 10.1109/DAPIC66097.2025.00066.

[37] A. Goswami and G. B. Shivaji, "Using big data and kafka to track resource utilization in real time," in *Proc. Int. Conf. Intell. Control, Comput. Commun. (IC3)*, Mathura, India, Feb. 2025, pp. 1028–1034, doi: 10.1109/IC363308.2025.10957433.

[38] Y. Jiao, J. Han, B. Xu, M. Xiao, B. Shen, and H. Sun, "Research on domain entity extraction in civil aviation safety," in *Proc. IEEE 3rd Int. Conf. Civil Aviation Saf. Inf. Technol. (ICCASIT)*, Changsha, China, Oct. 2021, pp. 384–388, doi: 10.1109/ICCASIT53235.2021.9633439.
[39] C. Wang, H. Sun, and Q. Chen, "Analysis of China civil aviation turbulence index eddy dissipation rate," in *Proc. IEEE 1st Int. Conf. Civil Aviation Saf. Inf. Technol. (ICCASIT)*, Kunming, China, Oct. 2019, pp. 607–610, doi: 10.1109/ICCASIT48058.2019.8972995.
[40] Q. Zhuang and T. Zhou, "International development of China civil aviation aircraft tracking and monitoring system," in *Proc. IEEE 2nd Int. Conf. Civil Aviation Saf. Inf. Technol. (ICCASIT)*, Weihai, China, Oct. 2020, pp. 1099–1103, doi: 10.1109/ICCASIT50869.2020.9368807.
[41] T. Ma and N. Chen, "Design of keyword extraction system for aviation safety information based on ArcGIS technology," in *Proc. IEEE 4th Int. Conf. Civil Aviation Saf. Inf. Technol. (ICCASIT)*, Oct. 2022, pp. 374–378, doi: 10.1109/ICCASIT55263.2022.9986831.
[42] M. Zhou, M. Liu, R. Zhang, and X. Wang, "Development status and suggestions for sustainable aviation biofuel," in *Proc. Int. Conf. Environ. Sci. Green Energy (ICESGE)*, Shenyang, China, Dec. 2022, pp. 187–191, doi: 10.1109/ICESGE56040.2022.10180375.
[43] Z. Wang and Y. Bai, "Prediction of international aviation carbon emissions and offsetting based on ICAO mechanisms," in *Proc. 5th Int. Conf. Power Energy Technol. (ICPET)*, Tianjin, China, Jul. 2023, pp. 1349–1353, doi: 10.1109/ICPET59380.2023.10367631.
[44] Y. Li, Y. Liu, T. Yuan, Y. Xu, N. Lei, and J. Ning, "Research on data governance methods and systems for large hub airports," in *Proc. 8th Int. Conf. Adv. Algorithms Control Eng. (ICAACE)*, Shanghai, China, Mar. 2025, pp. 1631–1634, doi: 10.1109/ICAACE65325.2025.11018979.
[45] A. Susanto, A. H. Fathulloh, Nuryasin, and A. Fitriyani, "Comparative analysis of key management service performance on AWS, Google cloud, and Oracle cloud with performance testing," in *Proc. 11th Int. Conf. Cyber IT Service Manage. (CITSM)*, Makassar, Indonesia, Nov. 2023, pp. 1–6, doi: 10.1109/CITSM60085.2023.10455569.
[46] O. Vuran, O. Akcin, M. Ravanbakhsh, B. Sankur, and B. Demir, "Deep learning driven content-based image time-series retrieval in remote sensing archives," in *Proc. IEEE Int. Geosci. Remote Sens. Symp.*, Jul. 2022, pp. 1940–1943, doi: 10.1109/IGARSS46834.2022.9884495.
[47] J. A. Shah and N. R. Iyer, "Building generative AI chatbot using Oracle cloud infrastructure," in *Proc. IEEE 15th Annu. Ubiquitous Comput., Electron. Mobile Commun. Conf. (UEMCON)*, Oct. 2024, pp. 79–84, doi: 10.1109/UEMCON62879.2024.10754774.
[48] I. Bojanova and A. Samba, "Analysis of cloud computing delivery architecture models," in *Proc. IEEE Workshops Int. Conf. Adv. Inf. Netw. Appl.*, Mar. 2011, pp. 453–458, doi: 10.1109/WAINA.2011.74.

**MICHAL KVET** (Senior Member, IEEE) became an Associate Professor of 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 text-books 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 the temporal databases, indexing, performance, analytics, and cloud computing. He strongly participates with Oracle Academy and is part of multiple Erasmus+ projects. Besides, he is the Consortium Leader of the Erasmus+ project dealing with the environmental analytics. He also organizes multiple database workshops annually.



**JARMILA ŠKRINÁROVÁ** received the M.S. degree in automation and control and the Ph.D. degree in technical cybernetics from Slovak Technical University, Slovakia, in 1986 and 2004, respectively. Since 2013, she has been an Associate Professor of computer science with the Department of Computer Science, Faculty of Natural Sciences, Matej Bel University, Banská Bystrica, Slovakia. Since 2015, she has been the Head of the High Performance Computing Centre and since 2023, she has been the author or the co-author of about 120 research articles, educational texts, and monographs. She often supports optimizations in scientific problem solutions by proposing appropriate optimization criteria and machine learning. Her research interests include parallel and distributed computing, and optimization and scheduling of tasks in cloud systems, varying from IoT tasks in edge computing to large-scale tasks in high performance computing. She is a long-time member of the program committee of IEEE conference Informatics. She is the long-time chair of the program committee of the international conference Didinfo, which focuses on didactic methods of teaching computer science in primary and secondary schools.