IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received February 22, 2022, accepted April 21, 2022, date of publication April 26, 2022, date of current version May 4, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3170711

The Complexity of the Data Retrieval Process Using the Proposed Index Extension

MICHAL KVET[1], (Member, IEEE), AND JOZEF PAPAN[2]
[1]Department of Informatics, Faculty of Management Science and Informatics, University of Žilina, 010 26 Žilina, Slovakia
[2]Department of Information Networks, Faculty of Management Science and Informatics, University of Žilina, 010 26 Žilina, Slovakia

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

This work was supported in part by the Erasmus+ Projects through the Project Cloud Computing for Digital Education Innovation under Grant 2020-1-HR01-KA226-HE-094713, in part by the Project Better Employability for Everyone with Oracle Application Express (APEX) under Grant 2021-1-SI01-KA220-HED-000032218, and in part by Increasing the Security of Communication Network Infrastructure, Grant system UNIZA, 2021.

ABSTRACT Data retrieval, access, and tuple identification are inevitable in database processing, ensuring performance. The entities and relationships form a relational database. Data themselves are not specifically formatted, requiring sequential data block scanning if the particular index is unavailable. This paper summarizes existing indexing principles focusing on the B+tree, which forms the default structure for data access based on the index key. Such design is reliable and prone to an increase in the amount of data. However, it cannot manage undefined values properly, whereas they cannot be mathematically compared. Secondly, migrated rows can be present due to the size demand extension after the update operation. Thirdly, the index is always balanced, resulting in additional demands of the transaction. All these factors are covered by the proposed paper, discussing the limitations, opportunities, and own solutions to improve the performance. Several architectures are discussed, maintained, and computationally studied, focusing on the size demands, processing time, and costs. By using proposed techniques, significant improvements can be reached. A pointer list is introduced for migrated row management to reference the index set from the data block and store reference path or using a data reflector. When dealing with transaction support, index management and rebalancing are shifted to the separate autonomous transaction. Thanks to that, the main transaction can be approved sooner with no reliability issues. Finally, the proposed paper introduces the NULL value management structure in the instance memory for the index node reference.

INDEX TERMS Index, migrated row, performance, post-indexing, relational database, transaction support, undefined values.

I. INTRODUCTION

The data world is continuously changing, significantly impacting proper data management. Over the decades, handling has been shifted from the data embedded systems through the file management up to the database complexity. Intelligent information systems point to the robust data to be covered. It is not now suitable to deal only with current valid data in a relational theory. Still, the definition should be open to access, manage and evaluate temporal databases, objects, JSON, XML, or texts stored in the large objects. All such data can be source for processing and stored and accessed by the database [1]–[3].

The relational database offers the stored data complexity, robustness, and security. The core element differentiating the database and file system is transaction support. Its management ensures the data to be reliable, passing all requirements and constraints. Transaction theory defines four aspects for dealing with databases – atomicity, consistency, isolation, and durability [4]–[6]. Atomicity ensures that the data operated inside the transactions (adding or changing the existing tuples) are either approved or refused totally. Thus, the transaction is treated as one inseparable element, which cannot be managed and evaluated partially. Consistency deals with integrity by emphasizing the constraints, which should be passed not later than the end of the transaction. Therefore, transaction shifts the database from one valid, consistent image to another consistent. The third aspect is isolation. Changes done inside the transaction are made visible after its approval by spreading the operations to the whole environment. Finally, durability covered by the logging ensures that the approved transaction changes are visible and will be

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

present in the system even after the system crash. Transaction definition automatically handles all these requirements by applying transaction logs (UNDO tablespace and REDO log files) and locks [7]. In many database systems, the emphasis is on shortening the transaction duration and number of the associated objects to the minimum. Such requirement is associated with the locking strategy, which limits parallelism. Generally, read and write locks are applied to the data to ensure consistency and access control. Database system Oracle, which is used as an evaluation environment, uses an improved technique where the read locks are not applied. Read consistency is done by the transaction log support building consistent data image valid either at the beginning point of the operation or the whole transaction. As a result, massive parallelism can be ensured, emphasizing log availability. Proper index and transaction management form the crucial elements to ensure the entire system's performance.

Shifting the data storage to the cloud environment, handling the data amount became almost unlimited. Data storage and CPU can be dynamically provisioned, optionally enclosed by the spread allocation [8], [9]. Transactions can be highly parallelized supervised by the internal data infrastructure optimization and scalability management to ensure performance. Then, the main optimization strategy of the general data management (local on-premise, cloud) should point to the data access perspective. Whereas the data amount and overall complexity are significantly rising, storing current valid data and the whole perspective of data evolution is inevitable. As a result, each object is delimited by the version evolving. Thus, identification of the object tuple is not made just by the unique identifier, and the time validity perspective should be covered, as well.

Data access performance is covered by the indexing strategy, strongly limiting the data amount to be loaded and evaluated. If the database index is unavailable or unsuitable for the query, sequential data scanning, represented by the block-by-block memory loading, is necessary. However, such activity can have a significant impact on performance. Each block must be loaded from the database (physical storage) to the memory in the first phase, followed by the parsing, tuple identification, and rows evaluation. The problem can cause block fragmentation, by which the loading efficiency is lowered [10], [11]. Block data amount can be significantly higher than optimal storing inside the blocks [12].

Moreover, data blocks are not created and associated with the table separately. Instead, the blocks are allocated at once, forming the data extent. As a result, even completely free blocks can be present and managed. Furthermore, even though the block does not hold any data, it is memory-loaded, whereas the system generally does not keep references to the empty blocks. Applying dynamics of the system by purging all data, the robustness of the update operations, and reliability aspects, the physical data repository can be strongly fragmented by increasing the size demands.

As evident, sequential data scanning is demanding and too time-consuming, pointing to the efficiency and performance of the whole process [13]. It is not related just to the data retrieval itself. Other operations like Update need to locate existing data tuples to be updated. Similarly, when adding new data tuples to the system (Insert statement), the system needs to check constraints (at least the uniqueness of the object identifier) by accessing the internal database structures. A database index as an optional structure associated with the table was introduced to reflect the performance issues. It is used for direct data row access. Generally, it can be defined by various structures, consisting of the ROWID value, used as a direct locator pointing to the database, file, block, and position inside the block. Using the index, sequential data scanning is reduced to the index search and block identification using ROWID [14], [15]. The finest granularity of the processing is the block itself, which must be fully loaded into the memory for the evaluation and consecutive processing. Thus, a proper database index can significantly influence the performance and reduce either processing time or CPU, I/O, and other system resources [8], [16]. The suitability of the index is crucial for the processing, whereas the order of the attributes delimits the index structure. Based on the [14], the order of the elements inside the index should reflect the query conditions and optionally values listed in the Select clause. If all required values (attributes and function calls related to the table) are not present in the index, using the ROWID values, the whole row can be easily located and memory-loaded. Vice versa, if the index is not in a non-suitable format, sequential scanning must be done, either covered by the scanning of the whole table or the index can be optionally used, as well, by reducing the amount of the data to be evaluated. As a result, multiple index sets are defined in the system to ensure performance.

On the other hand, index structures negatively impact the data modification operations (Insert, Update and Delete). In contrast, each change must be applied to the index to ensure the reliability and even usability of the indexes themselves. Consequently, there are two opposite streams - indexes improve the techniques of data retrieval, vice versa, data loading, and change operations have a negative impact on the performance due to incorporating necessity to the whole index set [17].

This paper deals with the relational databases supervised by the transaction robustness. Data structures and infrastructure are handled in section 2. Such a chapter is important for pointing to the consecutively proposed optimization techniques. Then, existing index structures and access methods are listed (section 3), pointing to the auto-indexing approach (section 4) available in the Oracle database system. It can manage defined index sets dynamically by evaluating benefits and costs over time, reflecting the workload. In principle, the existing index can be retained original, reconstructed, or marked for dropping. Complexity is always checked, whereas each index operation can influence the whole system. The main contribution of this paper is covered by section 5 by proposing own index architecture, by which the negative impacts on data change operations are lowered

or completely limited. Thanks to that, the range of the index set can be increased, but without a significant negative impact on the performance of data modifying operations.

To reach the complex architecture, section 5 is developed in multiple streams. Firstly, emphasis is taken on the block data architecture reflecting the data tuple storage. If the value precision is increased after any data change operation, tuple commonly requires additional space. This results in a lack of space in the data block in most cases. Basically, the new record will no longer fit in the original repository. Thus, the migrated row is created. In the original block, only the new address to the relevant block, where the data already resist, is stored. It results in degradation of the structure and an increase in processing costs. By default, we approach individual records, emphasizing the time and space spectrum through the created indexes. However, these indexes are not optimized for such change elements. Several solutions are discussed with emphasis on reliability, performance, and limitations. The best-obtained solution is to create an autonomous layer to identify migrated rows and incorporate changes into the structure without loading multiple blocks as is currently necessary.

The second addressed area is optimization at the processing changes in transactions. All changes are currently applied to the data layer and the indexes, which are always balanced in structure. The proposed solution creates a comprehensive index management environment outside the main transactions, emphasizing efficiency and immediate data accessibility. A monitoring system of the aircraft's position is used for the performance evaluation. Although the focus is on data safety, reliability, and correctness, the primary relevance is associated with continuous and immediate data availability, emphasizing the total costs, especially the processing time.

Finally, we deal with data reliability and undefined value identification and categorization, emphasizing the index structure set support.

II. DATABASE INFRASTRUCTURE

This section highlights the physical Oracle database infrastructure, whereas it looks different on various operating systems and differs from other database systems. The Oracle database system consists of the instance, database, and optimally, container and pluggable databases are covered and supervised by the real application cluster (RAC) environment [18]. A database is a collection of physical data files stored in the system. It is formed by the parameter files and control files supervising the infrastructure pointing to the data files repository. Data files hold the direct data in the block structure. Finally, the database is formed by the transaction and system logs and metadata stored physically. Vice versa, the instance is delimited by the background processes managing the instance and controlling the data access. Thus, the database is located on the physical disks, while the instance is delimited by the memory, which is shared across the individual connection sessions. It consists of various structures [19], [20]. The most relevant for this paper is Buffer cache - work area for executing SQL. Data cannot be updated directly in the database but must be memory-loaded in a block granularity for evaluation and processing. Log Buffer is a small staging area for maintaining transaction change vectors. A shared pool is the most complex consisting of dozens of subelements.

From the statement execution point of view, the most important type is the Library cache. It is a memory area for recently executed code in a parsed form. SQL statement processing consists of the parsing (syntactical and semantic check and Shared pool check to ensure that the SQL plan is not already processed and present in the memory structure). The optimization step is responsible for generating multiple execution plans. The query execution plan is obtained by the Row Source Generation. Finally, the statement is executed, either by the proposed SQL plan (hard parse) or by getting an already existing plan from the Shared pool (soft parse). As a result of the overall processing, the SQL plan is obtained (producing the set of operations covering the data retrieval process via index or direct data block loading [21], [22]), followed by the execution process itself.

There are two basic Oracle architectures - single and multitenant architecture [23].

Single-tenant was introduced in Oracle version 6 released in 1988 and was used until 2012. It uses a non-container database delimited by a one-to-one relationship between the instance and database in terms of metadata, Oracle data, and Oracle code.

The single-tenant RAC approach extends the definition by the clustered environment. Many instances simultaneously mount and open one database, which resides on a set of shared physical disks. All the instances share one database. RAC environment offers high availability, performance, scalability, and security ensured by the error-prone solution. A client connects to the Single Client Access Name (SCAN) RAC listener, which routes the traffic to the specific instance, ensuring the workload balancing across individual instances registered. Each instance node has a separate listener, processes, and memory.

In March 2017, the Oracle 12c version was created, offering new architecture [24]. A multitenant container database (root container database) contains physical data files covering the Oracle metadata. There are no user applications or code present. A pluggable database (PDB) is a set of data files that can be mounted dynamically during the process on demand. Thus, the data are separated in the pluggable form, whereas the Control, Parameter files, and Logs are in a core container. PDB is routed and started by one container at a time by indirect association with the instance. A client connects to the server by the listener, which creates a server process in the container instance. Then, the instance is connected to the container database forming the interconnection with the pluggable database. Thanks to that, databases can be attached and detached dynamically during the run. Moreover, compared to single tenancy, environment characteristics are stored just once for the whole container. Fig. 1 shows

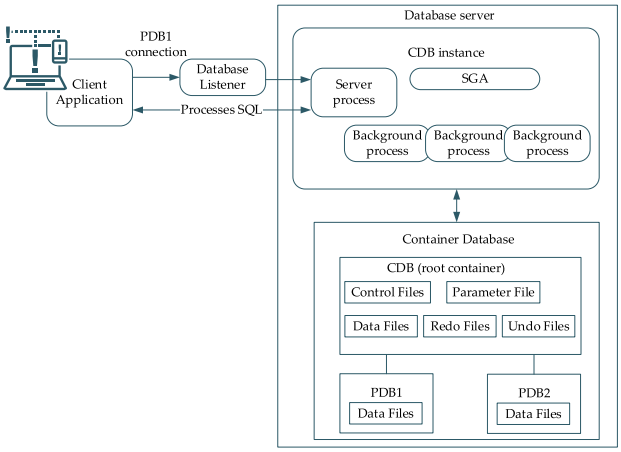

FIGURE 1. Container database architecture.

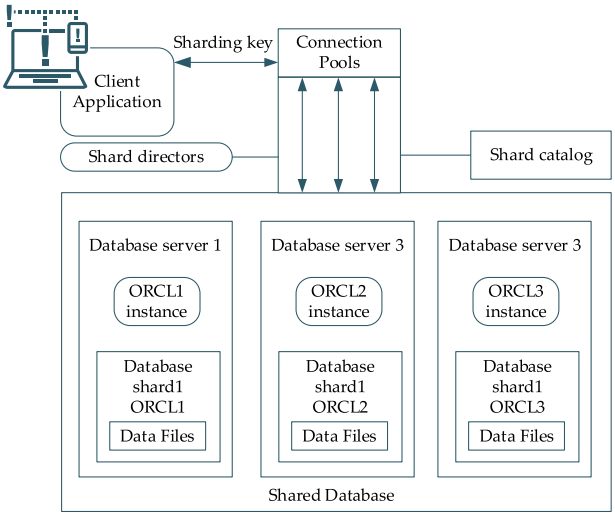

FIGURE 2. Sharded database.

the container database with two pluggable databases attached.

The general solution is provided by Multitenant RAC Database [18], where multiple node instances can be connected to one container database operating multiple pluggable databases. Compared with architecture presented in Fig. 1, there are numerous instances with separate memory and process structures. For each instance, SCAN listeners are connected to the individual listeners, evaluating the workload to ensure proper performance. This solution generally provides performance complexity, robust scalability, and error-prone management. Although the container does interconnect between the physical database and instance, if any instance fails, the workload manager automatically routes the survivors ensuring availability and reliability of the whole solution model.

From the performance perspective, the relevant model is also sharded database introduced in 2017 [25], providing linear scalability, fault tolerance, and geographic data distribution by using horizontal fragmentation across multiple regions [23], [26], [27]. It is shown in Fig. 2. Each partitioned database has own instance forming the sharded database, which is connected to the connection pool for the client connections supervised by the shard directors [25].

The architecture management and complexity have to be treated to cover the performance. More robust architectures have to be used to deal with the data distribution and availability domains to ensure reliability and continuous availability. Referencing the cloud environment, all databases are spread across multiple availability domains, operated by the RAC environment. Such architecture is also used in the performance study part of this paper.

III. INDEXING

A database index is an optional structure associated with the table by which the data can be easily located. Each index stores the particular attributes or function results, along with the pointers to the physical location of the row (ROWID) in the leaf layer [13], [15]. ROWID is a physical address of the row by pointing to the data file, block, and position of the row inside the block. Thanks to that, the data location operation is straightforward, and the processing costs can be significantly reduced for the data retrieval. Index management is part of the transaction. Therefore, it is always ensured that all data rows are accessible and identifiable by the index. Otherwise, the optimizer cannot use such a method for data management, whereas the reliability of the result set cannot be ensured. Although storage demands have risen utilizing the index, I/O operations, CPU, and memory resources can be significantly lowered. And finally, the most beneficial aspect is related to the time-consuming costs. Sequential data block loading, scanning, and evaluation are removed, replaced by the proper data block identifying where the relevant data reside. Whereas the index size is significantly smaller than the whole table [14], memory block loading necessity requires fewer blocks. Thus, the correct indexing strategy is crucial for achieving maximum database performance. From the user and server perspective, Fig. 3 shows the data query processing principles. The user is delimited by the user process of the client site, mapped to the server process of the database instance. For dedicated infrastructure, the mapping is one-to-one. Multiple client sessions can be routed to the common server process using a shared system infrastructure.

Fig. 3 shows the data flow between client and server. The user submits a query (step 1), which is routed to the Oracle server process (step 2) by notifying the optimizer to analyze the query (step 3) by selecting the optimized data access. Interconnection between the user and server is shown only schematically. In principle, various database architectures can be used, discussed in section 2. Steps 4 and 5 are the most crucial from the performance point of view. Such part is responsible for locating and transferring the data between the instance and the physical database. Finally, step 6 deals with

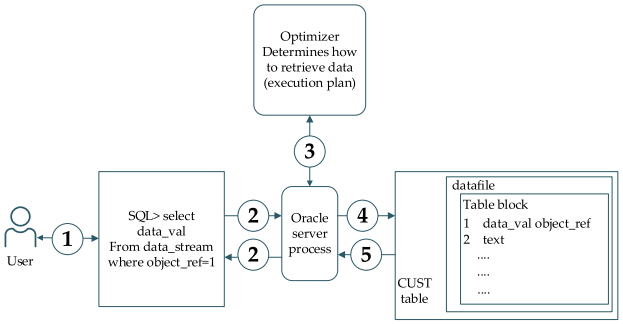

FIGURE 3. Query processing – client and server data flow.

the producing data result set related to the optimization techniques. Generally, two methods can be applied to optimize the data access [11]. The first approach is associated with minimizing the time to produce at least some data portion as a result set (SOME_ROWS hint). The second principle optimizes the whole result set consisting of all relevant data (ALL_ROWS hint) [7]. These parameter settings can impact the data access and indexes, whereas significant data amounts can be produced. Namely, the leaf pointer layer can have several ROWIDs for each non-unique index key value.

Three index types are currently used from the physical perspective – B-tree, hash index, and bitmap. In the following section, we will briefly summarize the principles, benefits and will point to the main limitations.

A. B+TREE

A B-tree index is a default balanced tree index suitable for the high cardinality columns or expressions. It consists of the root node, internal nodes covering the indexed values, and leaf nodes containing the ROWID pointers to the database structure. From Oracle 8i, pseudo column ROWID is delimited by 10 bytes. Prior releases reflected just 8 bytes. The main advantage of the access by the traverse path is associated with the balancing. The height from the root to any leaf node is always the same. Although it is commonly marked as B-tree, leaf layer nodes are interconnected, forming the bi-directional linked list. Data on the leaf layer are sorted based on the developed index format. Thus, the internal structure is B+tree instead of B-tree. One way or another, the B-tree indexing strategy follows robust improvement by accessing the data. Table 1 shows the reflection of the indexed data amount and depth of the index, calculated by the BLEVEL attribute of the USER_INDEXES data dictionary [24]. It omits the root element of the index. Therefore, the physical traverse path length is one greater. It compares the data row number (powers of the number 2) to the total depth:

Query

```
Select power(2,i), blevel + 1 as depth
 from user_indexes
  where index_name = 'STREAM_INDEX';
```

TABLE 1. Index depth correlation.

Number of records	Index depth
1	1
2	1
256	1
512	2
1024	2
262144	2
524288	3
1048576	3
134217728	3
268 435 456	4

From the above table, it is evident that the traversing using the index provides a powerful solution. Even if you have 300 million rows, just four index nodes must be accessed to reach the leaf layer consisting of the ROWID pointers. Although the consecutive number of data to be physically loaded is crucial, based on cost estimation, if the selectivity is too low, the system can prefer sequential scanning by assuming that a significant data amount will be obtained and composed as part of the result set [28].

B-tree index structure is a robust solution. It does not degrade over time, ensuring the whole table size efficiency. There are, however, three significant drawbacks – null value coverage, migrated row problem, and transaction support.

First of all, the B-tree index cannot cover undefined tuples. This is because they cannot be mathematically sorted; thus, locating such representation inside the index is impossible. As a result, generally, if the result set can contain such data, the index cannot be used, forcing the system to use sequential data scanning. For example, let have a table consisting of the temperature data obtained by the sensor. For simplicity, let obtained values be correlated just with the validity:

Algorithm

```
Create table temperature_tab
   (validity_date date primary key,
   temperature_value number);
```

Regardless of the data amount, sequential data scanning must always be used if you want to get values obtained by the specific time range, whereas undefined values can be present. Referencing the table structure, attribute temperature_value can hold a NULL value, whereas there is no limiting column constraint. Thus, although such a situation does not occur in

TABLE 2. Data amount perspective.

Data amount	Undefined values for temperature measurement	Unique values for temperature measurement
1 000 000	0	300 000

practice, the system cannot ensure it by the statistics, resulting in sequential data scanning.

Query

```
Select count(*) as ''Data amount'',
       count(temperature_value)
       as '' Defined values for temperature measurement'',
       count(distinct temperature_value) as
       ''Unique values for temperature measurement''
from temperature_tab;
```

Table 2 shows the above query result. The aim is to get unique values for a particular day. Firstly, all data are loaded to the instance memory block-by-block by extracting the VALIDITY_DATE. Then, the row falls into the time interval, covered by the consecutive step of hashing values to remove duplicates. If not, it is refused. The total costs for 1 million rows are 1 372. Fig. 4 shows the execution plan. Whereas generally, undefined values can be present, a table is sequentially scanned using the TABLE ACCESS FULL method.

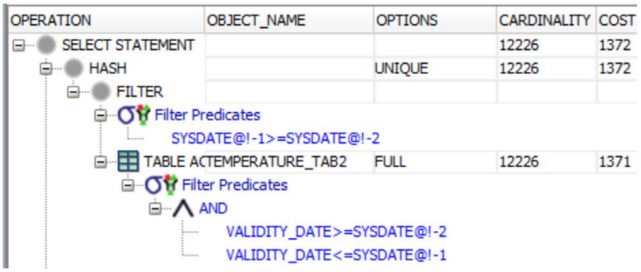

FIGURE 4. Execution plan handling NULLs – Table Access Full.

A range scan can be used properly by removing NULL handling from the result set (Fig. 5). It is ensured that all particular row are index referenced.

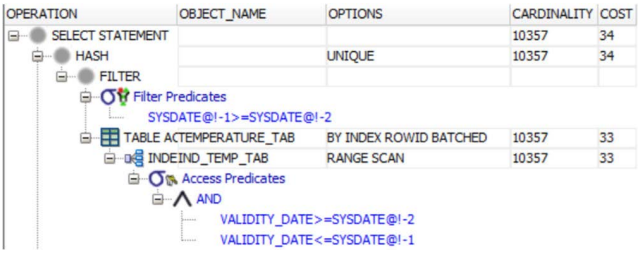

FIGURE 5. Execution plan - Removing NULL pointers from the query.

When sequential scanning is necessary, the total costs are 1 372. By removing NULL values, access can be done by the index method dropping the costs to the value 34, which reflects a 97,52% improvement. The index is prone to fragmentation and provides direct block access. Therefore, the data block amount necessary to be loaded is strongly limited. Moreover, usually, the index is already at least partially memory-loaded. Note that the size of the table is 39 936KB. Particular index storage demands are 19 456 KB.

However, systems must cover undefined values due to the data failures, delays, and reliability aspects [29]. Therefore, the segregation using the data layer is not feasible. Moreover, from the reliability management point of view, it is inevitable to detect anomalies, failures, and improper measurement.

Although it can be partially solved by the undefined (NULL) value transformation [30], a particular reference is a function encapsulated [31]. So, such definition must be used in the queries; otherwise, it would result in sequential data block scanning anyway. Moreover, function processing brings additional processing demands from the definition [32].

The second limitation is related to the physical database. As described in the previous section, files are fixed-size block-shaped. Thus, fragmentation across the blocks can be present. Therefore, the processing demands are extended by sequential scanning if the fragmentation is present. It can be partially solved by the blocking factor parameter of the database, by which the block is not filled completely. Instead, if the limit is reached, the new tuple record is stored in another block even if it still fits the block size. Such a principle is correct and does not significantly impact the performance using the index. Based on [18], [33], additional demands can be applied only in specific situations when the original data row could be located in a partially free block, which is currently loaded into the memory during the evaluation. It generally reflects less than 1% additional demands, reflected by the I/O block loading necessity [14]. The remaining free space in the blocks is used for the update operations. Therefore, the data row size can be extended during the change operation.

A typical example can be associated with the variable character or placing valid data value instead of NULL. As a result, the original data row is extended. If it fits the block space after the update, then it can be placed in the original position. If not, two approaches can be identified – it can be either relocated to a different block, or an overflow block can be applied [34], [35]. From the index access point of view, both solutions require additional data blocks to be handled. Namely, for overflow block, multiple blocks must be loaded by data accessing, even though the record itself could be stored in a single block. Therefore, if we want to access part of the row tuple in the overflow segment, we must read at least two blocks instead of one. However, by generalizing the solution, we can identify the direction of degradation of the system over time since the overflow block itself is only a variant of the standard block approach, so the size error can be present, resulting in a chain reaction. A more complex solution is done by relocating the data row to another block repository. As evident from Fig. 6 holding B+tree index structure, individual data can be accessed using the pointers from the leaf layer. However, the access direction is always from the index to the data. Thus, if the data position in the physical database infrastructure is changed, the migrated row is created. ROWID of the index points to the original block. So, data are not present, just the pointer to the correct block, where the row can be found [16].

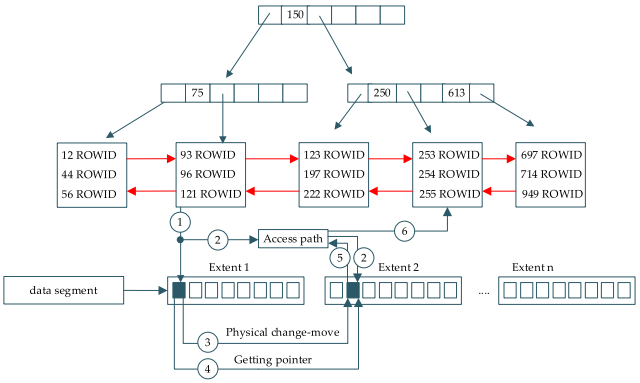

FIGURE 6. B+tree index highlighting the migrated row problem.

In [36], migrating row problem is treated by storing the used access path in a shared instance memory. If the migrating row is to be created and applied, the stored access path is used for applying the change, so the index locates the data in a new block. By using this technique, a particular index involves the change. Concluding, there is no complex solution, whereas several indexes can point to a specific row. Only one index covers the change. The rest indexes remain original. It can even cause one specific performance issue [33].

Fig. 6 shows the migrated row problem, as well. The physical data structure is defined by the data segment (table) and a list of extents with data blocks. Data migration occurs if the Update statement is to be done and the size of the row after the operation does not fit the original space. The system needs to find the suitable block to cover the tuple and apply the change by the following steps:

1. Getting data-position inside the block.
2. Storing existing access path inside the temporary storage (memory) – access path.
3. Changing the data block physically.
4. Getting pointer to the new block and interconnecting them in a directional manner (original -> new).
5. Applying the change using the temporary structure – access path.
6. Changing the original ROWID for the index.

As stated in [30], shared instance memory temporarily stores the access path to the data object, which can be used for index ROWID relocation to the new block, removing the impact of the data migration. A maximum of one index is used for each data access, over which the migrated row removal process can be applied. However, it should be noted that this operation is also an index-level change, so a transaction must encapsulate it to maintain consistency and restore the database after a crash [12]. A more important aspect is related to the fact that the original block still contains information about the object identifier that was in it in the past but was moved in the migration process. In principle, such information may be useless over time if all indexes have applied the migration removal process. From the database point of view, it cannot be identified, and therefore, the original block contains information that will never be used during the treatment [30], [37].

The third limitation is not directly related to the B-tree index but covers any developed index structure. Any relational database should always store reliable and accessible data. Thus, any data manipulation (DML) operation (Insert, Update and Delete) should handle not only the data themselves, but the index set should be taken into emphasis, as well. Thus, each operation is divided into two phases – applying change directly into the database (covered by the transaction logs) and pointing change to the index set associated with the table. Even after both operations are done successfully, a transaction can be committed. Vice versa, if any operation fails, the whole transaction must be refused. As evident, transaction definition is extended to cover the index set complexly. That can be, however, the bottleneck of the whole system. Therefore, the number of indexes must be balanced to optimize data access by the retrieval, but the other operations that maintain the index must be covered, as well. As a result, the number of indexes should be strictly limited to ensure the performance of the statements modifying data.

When dealing with the B-tree indexes, the defined limitation can be even more significant due to the index balancing necessity anytime. Namely, the index height should be the same for any leaf node after each operation. Consequently, transaction management is mostly devoted to index management, not the manipulated data themselves.

The following section provides a brief analysis of conventional index management during the Insert operation. For the evaluation, a flight data model will be used [36], [38], [39] covering airspace definitions in a temporal manner, surrounded by the planned and real route, supervised by the flight points (positions of the airplane) - intended and real. First, each airspace was delimited by the unique character string name, numerical sequence number, validity time frame, and airspace parameters, like minimal, maximal flight levels. Next, each route was identified by the ECTRL ID, sortable by the SEQUENCE NUMBER by entering and exiting particular airspace by the positional data – LONGITUDE, LATITUDE, and FLIGHT LEVEL. Finally, each flight was monitored by trajectory, speed, height, and other parameters. Particular data were range partitioned over the year quarters covering the flights from 2015. A deeper data set description can be found in [28], [39].

B. HASH INDEX

Hash index is formed by the array of buckets to which individual rows are associated. Each bucket stores a list of pointers to the individual rows which belong to it. Hash indexes are based on hash functions, which map the index key (K) to one bucket. Each bucket (array element) stores only the pointer, not the hashed values. Generally, key-value can be variable length. A hash function is responsible for mapping these key values to the fixed structure. Therefore, it must be deterministic and distribute data as uniformly as possible [37]. Fig. 7 shows the principle of using hash index during the data evaluation.

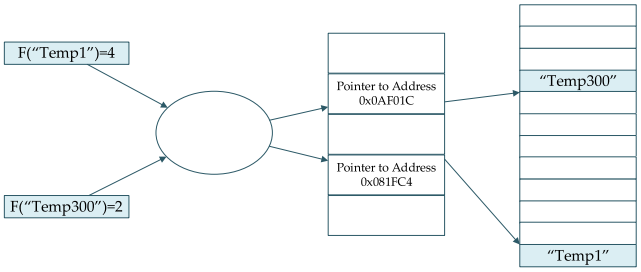

FIGURE 7. Hash index.

Obtained data are shifted to the hash function as an input resulting in assigning bucket. It is then used for storing a pointer to the particular input row.

Similarly, such an approach is also used for any operation and data retrieval. As visible, a crucial element influencing the efficiency of the processing is just the hash mapping function, which must be robust for any data, small data set, and large data amount simultaneously. Moreover, data value patterns can evolve, and such functions must be aware of it to ensure uniform distribution anytime. Finally, an important factor is related to array sizing. Whereas each bucket stores the data pointers randomly, all of the referenced blocks must be loaded into the memory for consecutive evaluation and condition checking [40].

A gradual reduction of hash indexes during the last period can be perceived. This circumstance is due to an increase in the amount of data that must be evaluated by shifting the complexity. Currently, systems do not cover just current valid data but the whole spectrum over time [41], [42]. Therefore, it degrades the definition of the hash function, which fails to protect the ever-changing structure and index values. In the past, the hash function was evaluated from time to time and possibly replaced by a newer improved version, but it would be almost a daily routine with the increase in the amount of data [15].

The second problem is the dimensioning of the field structure itself in size. Data should be uniformly distributed across the buckets, covering the amount changes. Moving to autonomous cloud structures makes it possible to provide transaction management and data archives, where outdated data from the online structure are gradually transferred [5]. Thus, the amount and properties of the data rise can significantly change the characteristics over time. In the worst-case scenario, processing may degrade to the need to process all data if a bucket overflows. In this case, in contrast to sequential processing, it would be necessary to calculate a hash and build an index, which would be used to process most of the original data anyway.

C. BITMAP INDEX

A bitmap index is a specific database index, mostly used in data warehouses to reduce response time for large queries. Its main benefit can be reached for low-cardinality index columns with a high number of table tuples. Thus, it is used mostly for columns with a significant amount of duplicate values. In comparison with other indexes, dynamic storage requirement reduction can be identified. Due to binary operation processing with a small number of CPUs, dramatic performance gains are present. The bitmap index structure is very effective for the analytical query processing by applying binary operations, which are very fast. Vice versa, bitmaps do not focus on any data change. Therefore, its reconstruction during the change operations is strongly demanding.

In contrast, the whole structure must be rebuilt if a new value to be indexed (which is not already present in the index) is added [40]. Fig. 8 shows the architecture of the bitmap index by using just binary values for the internal representation. Each bit of the bitmap corresponds to one ROWID; thus, if the bit is set, it means that the referenced data tuple contains the key value. The massive compression of the data inside the index can be applied. Individual rows are processed by the binary operation merging for the data retrieval.

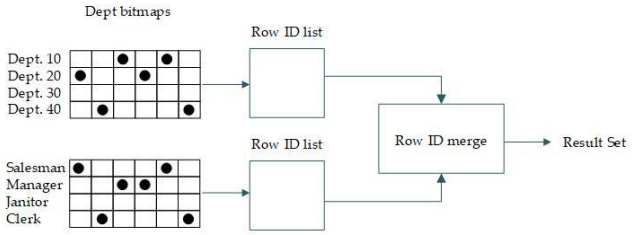

FIGURE 8. Architecture of Bitmap index.

Such index type is not suitable for online transaction data processing, whereas it is characteristic of data change operations online. Therefore, the bitmap index will not be evaluated later for the performance evaluation. Performance studies applied to autonomous data warehouses can be found in [43]–[45].

D. PARALLEL & NOLOGGING CLAUSE

Creating an index offers you various clauses and options that can significantly influence the management and inner performance. In this section, two extensions are highlighted – parallelism and transaction logging.

The parallel option allows the system to allocate multiple index and table scanning processes. It is associated with the index definition and is mostly used during index creation. The creation process is preceded by the whole table scanning, identifying the rows, and extracting their positions – ROWIDs. Thus, the table is full-scanned, up to the last associated block (delimited by the High Water Mark [37]). Using the parallel option, the table block set is divided into several sub-parts, each assigned to one process run in a parallel mode. Therefore, an index can be created rapidly sooner, depending on the number of CPUs, physical data definition, disc storage allocation, configuration, distribution, and partitioning [26], [46], [47]. The following code highlights the description of parallelism. Based on [48], the parallel option

should be set to the value (n-1), where (n) represents the total amount of allocated CPUs for the database instance. One core is then responsible for the supervision and workload division.

Algorithm

```
Create index index_name
    on table_name (list_of_attributes)
      parallel n-1;
```

Although such a clause is mostly related to the creation process, its significance can be identified during the data retrieval and row search. Whereas the index key is not commonly used, multiple ROWIDs can be located by accessing the index leaf layer, which can be present in multiple data files. The data block loading process can be done parallelly [49]. No matter how the data are processed, any change at the index level must be covered by the transaction [24]. The transaction aims to transfer a consistent database state to the new one by applying all the integrity definitions and constraints. Before transaction approval, any change on the data must also be applied to the relevant index set. As stated, index operations are covered by the transaction. Any modification is logged in the UNDO or online REDO logs to ensure recovery possibility [7]. In case of any data error, the restore and recovery process comes to the scene. Logs ensure the consistency and durability of the approved transaction, even after the failure. Then, the backup is taken, followed by logs extractions and changes re-execution. Transaction logging is unnecessary when dealing with the index, whereas the index itself does not hold any additional information. It just creates specific access optimized layer. It cannot happen that the index refers to data that is not yet in the database or applies changes that have not yet been made.

The introduced NOLOGGING clause allows you to skip to the transaction logging process related to the index [37]. In the past, it was represented by the RECOVERABLE and UNRECOVERABLE clauses. However, by introducing partitioned tables and LOB storage, particular characteristics are useless. Therefore, the NOLOGGING clause of the index instructs the transaction manager process to omit the REDO log stream. However, UNDO must still be used, whereas the transaction can be refused, so the index must be returned to the original form. Vice versa, REDO logs are used only in case of data or server failure, like disc errors, hardware, or unplanned electricity outage, which end with the instance disconnection and server shutdown. During the starting process, the system recreates the structures to the point immediately before the failure. Backups and online and archive logs are needed [50]. Whereas for the index, not all instructions are present. The particular index must be marked as UNUSABLE. In that case, it must be completely reconstructed by the REBUILD operation [50].

On the other hand, it can provide significant performance benefits during the index creation process and any changed operation. The NOLOGGING clause assumes that the hardware and individual network and electricity sources are reliable and that failure probability is significantly low.

E. CARDINALITY & SELECTIVITY

Indexes play a key role in high cardinality tables with multiple data blocks. If the table consists of just a few blocks, an index does not significantly impact and benefit. In such a case, index treatment and its block loading necessity are not balanced by reducing table block evaluation perspective. Rising the data heterogeneity, changing frequency, and re-loading data to the separate analytical or warehouse repository, the importance of the index definition, impact, and necessity is strongly increasing.

F. SEQUENTIAL SCANNING

Access to data through the index brings a significant increase in performance. Index traversing is straightforward and balanced and accepts all core data structure rules. At the bottom layer, the address of the record can be gotten, which is the direct and fastest way to access a particular record. An opposite way for data access is sequential block scanning. Fig. 9 shows the principle. Each associated data block (up to the last block marked by the High Water Mark symbol) is taken to the memory by the I/O operation. Afterward, it is parsed to extract the row. Then, it is evaluated whether it passes all query conditions (represented by the Where clause of the Select statement). Finally, it is shaped based on the Select clause focusing on the attribute or function results and interconnected with the existing partial result set. The data block loading process is demanding due to several reasons. Firstly, it is loaded into the memory buffer cache, where the free space must be identified (clean blocks). If there is no space, a particular area must be freed by transferring dirty block elements to the database, represented by the I/O operation of the block and transaction log. Thus, an additional database loading operation is added. Secondly, particular block content is crucial. Each data block is related to just one table. However, generally, it does not need to be full. Fragmentation of the blocks is a relevant issue influencing the performance of the sequential scanning – Table Access Full. Finally, blocks associated with the table are not allocated separately but are encapsulated by the extent – set of blocks. As a result, in the initial processing phases, some of them can even be totally empty, but covered by the upper limit – High Water Mark.

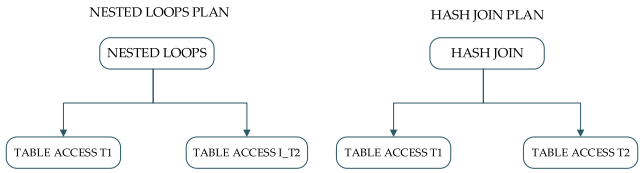

FIGURE 9. Dynamic access plan routing – architecture.

G. INDEX ACCESS PATH TAXONOMY

Access path selection is one of the most important elements of the optimizer decision. It significantly influences the process and principles of data retrieval. Generally, two basic types of access paths can be identified – Full Table Scan (Table Access Full), which is used, if there is a necessity to scan and retrieve a large portion (or even the whole structure) of the table data. Another situation covering such a technique involves accessing a small table with just a few blocks. The above perspectives are optimized and do not bring performance issues, whereas pointing to the index would bring additional demands, which are not balanced by the benefits [28]. In addition, index data loading and traversing require I/O operations. In the final stage, original data blocks must be memory transferred and evaluated.

A different strategy is associated with the index access methods. The index obtains relevant block identification, providing a ROWID set (or direct data, if all required are already part of the index). For decision making, selectivity, statistics, and estimated cardinality are important. Selectivity (S) of the index (I) is the number of distinct values (d) contained in the data set, divided by the total number of records (n). (A) represents the treated data elements [51]:

Algorithm

```
S(I) = d / n
        d => select count(distinct A) from T;
          n => select count(*) from T;
```

Current statistics ensure optimal input set for the optimizer decision making as a framework view of the data, focusing on the cardinality, column selectivity, ranges, undefined values coverage, etc. Statistics are generated either automatically during the maintenance windows or on-demand by the following method calls of the DBMS_STATS package:

Algorithm

```
DBMS_STATS.GATHER_INDEX_STATS
  -- Index statistics
DBMS_STATS.GATHER_TABLE_STATS
  -- Table, column, and index statistics
DBMS_STATS.GATHER_SCHEMA_STATS
  -- Statistics for all objects in a schema
DBMS_STATS.GATHER_DATABASE_STATS
  -- Statistics for all objects in a database
DBMS_STATS.GATHER_SYSTEM_STATS
  -- CPU and I/O statistics for the system
```

H. INDEX ACCESS PATHS

Index scan performs optimization sorts by the index strategy getting the key and ROWID pairs. Such ROWIDs are then used for the block location [17]. A special case is covered if only indexed values are accessed. In that case, where no additional blocks are necessary to be loaded, processing ends and ROWIDs are not used. Vice versa, if additional column values are required, Table Access by Index ROWID method is used [52]. The following techniques can be classified [37], [52]:

- Index Unique Scan method is based on unique conditions placed in the Where clause ensuring that no more than one row is obtained.
- The Index Range Scan method is based either on the range condition or generally based on any condition covered by the index. Therefore, there is no guarantee a single record will be provided as a result.
- Full Scans – Index full scan methods cover the set of operations, which read the whole index entries. Thus, it is an analogy to the Full Table Scan method, but it benefits because the result set has already applied a sort operation (Index Full Scan). If all required data are part of the index, but the index shape is unsuitable, the Index Fast Full Scan method can be used. It is based on the index size being always smaller than the referenced table. Moreover, an index is more precisely optimized from the storage perspective.
- Index Skip Scan has been introduced on the Oracle 9i version [24], [40]. It skips the leading column of the composite index by using the prerequisite of low selectivity of the first index column and high selectivity of the rest ones.

I. PLAN TRACING

The above section focused on access methods for just one table. Naturally, the query is commonly complex, joining multiple tables. In that case, access method selection depends on the foreign key definition, indexing strategy (if any), cardinality, etc. [14]. Generally, three core methods can be located. A Nested Loop is a simple method that presorts the child table (with a foreign key) in the first phase. It, therefore, uses the fact that foreign key is not indexed. It is the most demanding Join operation. Merge Join is more straightforward, whereas both sets are indexed, thus can be treated as sorted sets. One pointer is used for each set by using merge operation. Nested Loop method limitation is associated with the sorting necessity in the first phase, which can be time-consuming for large tables. The Hash Match method does not sort the data but uses a hash function to divide data into small buckets then treat it in parallel. Each bucket then stores a relatively small amount of data, so the joining operation is easier, either by sorting such bucket data or scanning it fully [37]. Nested Loop and Hash Match methods strictly depend on the quality and accuracy of the statistics. The optimizer must select the appropriate form based on the table and result set cardinality estimation. If they are not relevant, performance can degrade, reflecting the wrong decision. To limit such a problem, an additional optimization method was introduced in 2013 by DBS Oracle. In 2017, it was implemented in SQL Server, as well. Adaptive join can dynamically shift the processing method based on the threshold at run time [37].

In principle, two data plans are created (Fig. 9). Two server parameters secure definition and management. The original form of the Optimizer_Adaptive_Features has been marked obsolete in Oracle 12.2 [37]:

Algorithm

Alter {session | system}
set **Optimizer_Adaptive_Features** = {true | false}
[scope = both];

Fig. 10 shows the dynamic execution plan. Rows marked by „-“ are inactive. Thus, Nested Loop is used, whereas it requires lower processing demands. The total costs are 3. By using Hash Match, total demands are 5, whereas the Hash structure has to be created by calculating data positions in hash buckets.

```
PLAN_TABLE_OUTPUT
SQL_ID  4r3harjun4dvz, child number 0
-------------------------------------
SELECT a.data AS tab1_data,       b.data AS tab2_data FROM   tab1 a
    JOIN tab2 b ON b.tab1_id = a.id WHERE  a.code = 'ONE'

Plan hash value: 2672205743

-----------------------------------------------------------------------------------------------
|   Id  | Operation                            | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|     0 | SELECT STATEMENT                     |               |       |       |     3 (100)|          |
|-  *  1 |  HASH JOIN                           |               |    25 |   425 |     3   (0)| 00:00:01 |
|     2 |   NESTED LOOPS                       |               |    25 |   425 |     3   (0)| 00:00:01 |
|     3 |    NESTED LOOPS                      |               |    25 |   425 |     3   (0)| 00:00:01 |
|-    4 |     STATISTICS COLLECTOR             |               |       |       |            |          |
|     5 |      TABLE ACCESS BY INDEX ROWID BATCHED| TAB1       |     1 |    11 |     2   (0)| 00:00:01 |
|  *  6 |       INDEX RANGE SCAN               | TAB1_CODE     |     1 |       |     1   (0)| 00:00:01 |
|  *  7 |     INDEX RANGE SCAN                 | TAB2_TAB1_FKI |    25 |       |     0   (0)|          |
|     8 |    TABLE ACCESS BY INDEX ROWID       | TAB2          |    25 |   150 |     1   (0)| 00:00:01 |
|-    9 |   TABLE ACCESS FULL                  | TAB2          |    25 |   150 |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("B"."TAB1_ID"="A"."ID")
   6 - access("A"."CODE"='ONE')
   7 - access("B"."TAB1_ID"="A"."ID")

Note
-----
   - this is an adaptive plan (rows marked '-' are inactive)
```

FIGURE 10. Dynamic execution plan of the query – table joining.

IV. AUTO-INDEXING

Automatic indexing was proposed in February 2019. It is now available just for the DBS Oracle, removing the database administrator intervention necessity for index management and structure optimization. Using artificial intelligence, machine learning techniques, and complex analytics, the database manager automatically evaluates the need and impact of the index set to ensure complex performance. It does not only create new indexes, but the existing ones are emphasized and optimized to guarantee topicality, like transforming the single-column index to the concatenated version, etc. [18]. The SYS_AI name denotation marks Auto-created indexes as a prefix [53]. The automatic indexing feature makes the index invisible to ensure performance and no existing query degradation. Generally, automatic indexing can be set using two modes – REPORT_ONLY or implemented characterizing the output of the evaluation analysis, which can contain either a new index set implementation (IMPLEMENT option) or just a set of hints is provided (REPORT_ONLY). Then, the administrator is then responsible for the implementation itself.

Algorithm

dbms_auto_index.configure (’AUTO_INDEX_MODE’,
{’IMPLEMENT’ | ‘REPORT_ONLY’ });

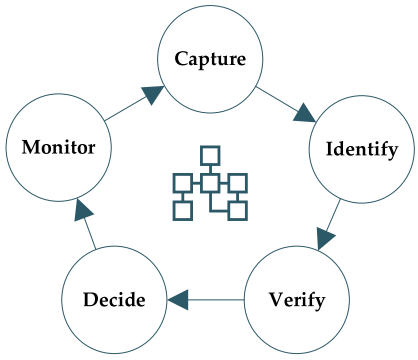

FIGURE 11. Auto-indexing management staged process.

Fig. 11 shows the staged process of the automatic indexing. Firstly, it analyzes the current workload by extracting processed queries across all data manipulation layers (Insert, Update, Delete and Select). Although the data retrieval is emphasized, other operations should not provide significant degradation, and the overall performance should benefit, respectively (CAPTURE phase). Then, operations for the optimization are extracted (IDENTIFY phase) followed by creating new indexes, respectively optimizing existing. All newly created indexes are marked as invisible to evaluate costs. If the benefits are positive and significant, during the DECIDE phase, the system implements indexes and makes them usable by transforming the changes to the visible form. Finally, existing indexes are monitored, whereas the workload, data access methods, principles, and patterns can evolve.

Query

Select index_name, visibility
from **user_indexes**
where index_name like ‘**SYS_AI**%’;

Fig. 11 shows the automatic indexing management process. First, the index is created as invisible, followed by checking all SQL statement execution plans. If all statements degrade, a particular index will remain invisible and is marked for dropping. Vice versa, if all statements benefit, the specific index is made visible and accessible during the optimization. However, the most likely scenario is reflected by the state that some statements show benefits, the rest perform degradation. In that case, the index is made visible, but the optimizer creates a SQL plan baseline to prevent existing queries from being degraded [36], [54]. Thanks to that, there is no performance regression done by the automatic tasks – analyzers and advisors. Fig. 12 shows the list of advisors supervising the auto-indexing process.

Query

```
Select *
  from dba_advisor_tasks
    where owner = 'SYS'
      order by task_id;
```

```
TASK_ID TASK_NAME                    ADVISOR NAME
      2 SYS_AUTO_SPM_EVOLVE_TASK     SPM Evolve Advisor
      3 SYS_AI_SPM_EVOLVE_TASK       SPM Evolve Advisor
      4 SYS_AI_VERIFY_TASK           SQL Performance Analyzer
      5 SYS_AUTO_INDEX_TASK          SQL Access Advisor
      6 AUTO_STATS_ADVISOR_TASK      Statistics Advisor
      7 INDIVIDUAL_STATS_ADVISOR_TASK Statistics Advisor
```

FIGURE 12. Auto-indexing advisor list.

For the automatic index configuration, a separate index tablespace can be created. Based on the [33], data and index extraction can significantly impact performance, whereas access to the index and data can be done in parallel. Moreover, indexes have different demands, so the index block size can be smaller to focus on the relevant traverse path [55].

Algorithm

```
dbms_auto_index.configure
  ('AUTO_INDEX_DEFAULT_TABLESPACE',
  'TBSPC_AI');
dbms_auto_index.configure
  ('AUTO_INDEX_EXCLUDE_SCHEMA',
  'FLIGHT_DESC');
```

Indexing strategy can be monitored and parametrized using DBMS_AUTO_INDEX package methods.

Another optimization strategy was introduced in Oracle 12c by focusing on the In-memory column store. In that case, the traditional row format is replaced by the column definition in the instance memory (located in a System Global Area). It applies to the large tables, which would originally perform fast full scans. The principles are associated with the requirement to locate just a small column subset [37].

The columnar memory format option is supervised by the Inmemory_size system parameter, which is static, and therefore, instance restart is needed to be applied. Size cannot be less than 100 MB [11], [56].

Algorithm

```
Alter system set
 inmemory_size = 500m
   scope = spfile;
```

A. SUMMARY – BENEFITS AND COSTS, RESEARCH PERSPECTIVES

Indexing is a relevant, robust, and complex issue concerning database access optimization. Several index approaches have been discussed in the previous section, mostly focusing on the common, widespread techniques based on the B+trees. Such data structure is not only the leading solution in a puzzle of performance guarantee but produces a robust solution in terms of data evaluation – either in the size perspective but also in structure change management. It is also error-prone and ensures access fragmentation reduction. Reflecting the architecture, the issue of migrated rows can be identified, present in case of change operations, where the new data tuple does not fit the original block dimension. Moreover, indexes highlight the pointers to the data. Thus, if the physical storage is changed on the data file or tablespace manner, the whole index is marked as unusable and must be completely rebuilt to make it usable.

The second aspect highlighting the architecture of the B+tree index is the physical representation. The B+tree structure is always balanced, so the height from the root to any leaf node is always the same. It must be done in case of any change at the index. It can be done at the record level (default solution) or the entire operation (APPEND hint). When multiple operations are done in parallel, the balancing process can be demanding and long-lasting.

The third optimization strategy focuses on the transactions. It is ensured that the index is correct by corresponding to the current data state. The transaction cannot be approved without applying all changes to the whole index set. Moreover, to ensure the recovery, any change done on the index level is stored in the transaction logs, stored physically in the database storage of online Redo log files. Such an option has two perspectives. Firstly, the index can be reconstructed to a consistent state after the instance failure and remain usable. Even after the storage failure, in the event of disc corruption at the index block level, the particular structure can be restored and recovered from the backup, followed by applying logs. Thus, the structure is robust and error-prone.

On the other hand, the second perspective is based on the additional demands. To ensure the recovery process, all logs between the timepoint of the backup and the current time must be present. Otherwise, although the recovery process starts, it raises the incompleteness of data – discontinuity of the system change number. The whole recovery is corrupted, resulting in marking relevant indexes as unusable. Thus, index transaction management, while ensuring correctness, if the ability to archive all logs is not enabled, the ability to recover completely is lost. Moreover, parsing many transaction logs can be very time and resource-consuming.

Finally, multiple access methods were introduced and discussed over the decades. Data identification and location can be made by sequential scanning or by using indexes. Decision-making is done by the database optimizer, pointing to the current statistics. However, they are not updated automatically, but their refreshing operations are planned to the maintenance windows [40]. When the robust data stream is present and data evolve rapidly, even currently completed refresh is no longer relevant. It may result in improper decision-making, whereas the input data are incorrect. Optimizer management is based on heuristics, influencing the

selection of access methods. Therefore, it is necessary to focus on multiple aspects concerning the execution plan. In this paper, we propose new techniques to ensure up-to-date statistics. Several execution plans are checked if the data pattern or amount is changed significantly. Generally, an already calculated plan is used, if available directly. However, in our proposed solution, the definition is extended by summarizing the data structure, data amount, and reflection perspective. Thanks to that, the database system can autonomously evaluate the existing SQL plan, emphasizing the current data image.

Although indexes form robust solutions to ensure performance, various improvement streams can be identified. In this paper, we focus on the following segments by proposing our techniques:

- migrated rows identification and reflection,
- index automatic balancing,
- index structure efficiency evaluation and consideration,
- index management outside the main transaction,
- control of undefined values (NULL),
- relevant data block identification,
- dynamic execution plan optimization.

V. OWN SOLUTION

Section 4 summarizes existing database approaches to deal with the indexes by pointing to various research spheres. In the last paragraph, there is a definition of the techniques covered by our research. The solution is based on the B+tree index as a core element, extended by various other data structures and proposed background processes to handle it. The whole section is divided into multiple parts to focus on different optimization strategies. In the conclusion section, a discussion about the limitations and future research perspectives is present. By reaching the complexity of the proposed architecture, significant performance improvements can be identified.

A. MIGRATED ROWS IDENTIFICATION AND REFLECTION

The database is defined as a set of data files formed in the block granularity. The interconnection between the memory and database is a tablespace, which delimits the block size. Migrated rows are created if the original data record no longer fits into the original block after the change. The system physically searches for the new repository by locating the block to handle a particular row. If there is no space available, a new extent (set of blocks) is allocated. Thus, in principle, there is no problem finding the new location. However, it can have an impact on the indexes. There is no specific pointer of the opposite direction - definition from the block to the relevant indexes. It would be necessary to scan all indexes to apply migrated row, which is represented by adding a new pointer from the original one to the new repository. As a result, multiple data blocks need to be loaded to locate the row using the index. In general, it does not have to be just two blocks - the original and the new block storage, but the structure may be larger, depending on the overall operability of the table. Thus, the migration of one row can be spread across multiple blocks, all of them in the chain must be loaded sequentially, and just the last one is finally required. Thus, if the update and delete operation frequency is high, performance can significantly degrade over time. Currently, it can be solved just by rebuilding the index completely. Reflecting the input stream (operated by various data manipulation operations) and data evolution, such an approach is unsuitable, and it is necessary to apply and limit migrations online dynamically. We propose and discuss different solutions to analyze the impact and evaluate benefits. As evident, the core element is to identify migrated rows and remove additional pointers to ensure that the existing index set always reflects the current blocks. From the architectural perspective, three solutions are proposed.

B. MIGRATED ROWS - SOLUTION 1

The first solution limiting the impact of migrations is based on the data block structure extension. A list of pointers to the index set is present for each data row. Thanks to that, it is easy to locate any index. It points to the particular leaf block, which holds the ROWID reference. Thus, the original ROWID value is replaced by the new address, where the data reside. From the physical point of view, a new dynamic array is allocated inside the main block (Fig. 13). As evident, if a new index is created, a new array element must be used to reference the index.

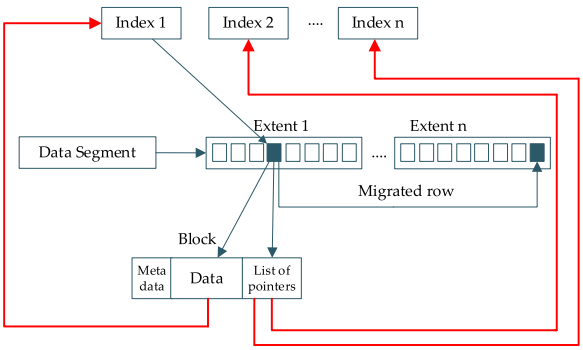

FIGURE 13. Architecture of solution 1 – list of pointer structure.

Therefore, each operation is applied at the index level and at the extended structure (List of pointers - address field) to eliminate migrated rows. As a result, it is necessary to load and update the block itself during any change operation. In addition, if the original index were set as NOLOGGING after the instance failure, it would be necessary to reconstruct the entire index and all blocks, which can be significantly expensive. Mean Time To Recovery parameter limitation cannot be reached – indexes will be unusable, and address fields will become invalid. If the automatic indexing is enabled, the dynamic structure would have to emphasize the efficiency of address field expansion. Concluding, adding an index could create another migrated row necessity, not at the level of the

data themselves, but at the address field granularity. Block itself does not hold only data themselves, resulting in the data block amount extension. Sequential scanning would bring additional demands when dealing with the indexes. Originally, several rows could be present inside one block. For now, data representation and available size are lowered. Thus, using an index can increase processing costs and the number of I/O operations.

Moreover, suppose the number of indexes was too large. In that case, the structure could not fit in the block itself. An overflow block would have to be defined, which would cause additional costs for record processing, as the entire overflow must be loaded whenever the data changes (migration).

C. MIGRATED ROWS - SOLUTION 2

Solution 1 is based on the main block structure extension. The original segment for data holding is divided into two modules holding the data, and the address field module is used inline. The second solution principles are the same, but the address field is extracted into a separate data structure, also block-oriented. These address fields are tree-structured, so more efficiently can be searched. Each element has an object identifier, extended by the physical data position (ROWID) and reference list to the indexes. Such reference list can be stored either inline inside the index (SOLUTION 2a) or out-of-line in a separate block array (SOLUTION 2b). The advantage of SOLUTION 2a is based on the assumption that any node and particular reference list is in the same block. However, the implementation is not robust from the perspective of adding or removing existing indexes (done, e.g., by automatic indexing). Vice versa, index set extension or reflection of existing index approaches benefit if the reference list is stored separately. In that case, a particular reference array can be extended at any time. However, the number of blocks to be loaded is increased by one for any data row. Conversely, reference list extension can be located in the overflow block structure associated with each node reference list in case of block fulfillment. Fig. 14 shows the SOLUTION 2a architecture.

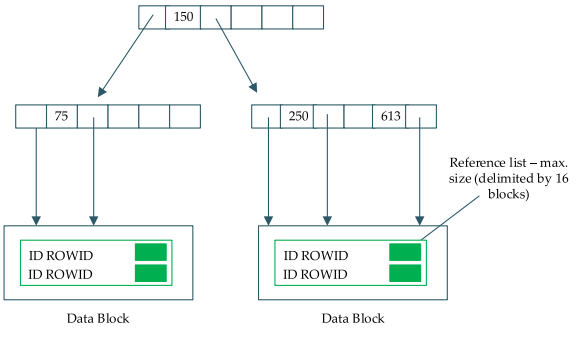

FIGURE 14. Architecture of solution 2a – inline reference list.

The architecture of SOLUTION 2b is shown in Fig. 15. Note that the overflow structure is optional and added dynamically if one block cannot cover the reference list completely.

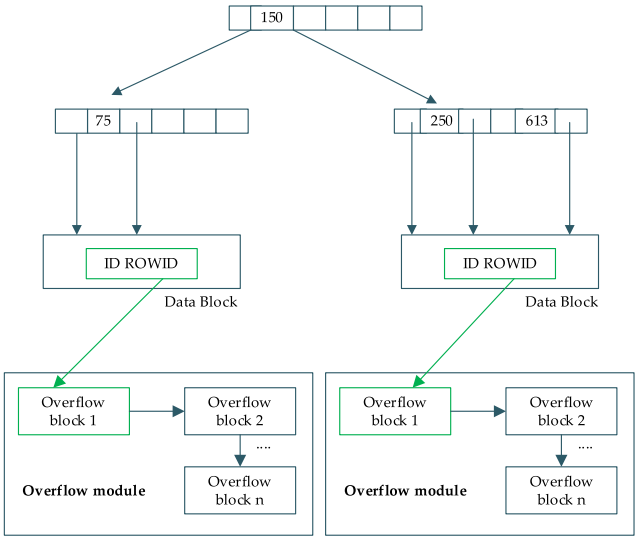

FIGURE 15. Architecture of solution 2b.

The introduced Index_referencer background process is responsible for the whole structure management. Its workers (Index_referencer_worker(n)) are responsible for overflow segment optimization by calculating the block usage – blocks are split and merged autonomously to ensure performance. Index_referencer_worker background process is delimited by the worker identification, followed by the ID represented as „n“ in a numeric format.

D. SOLUTION 3 – MIGRATION-POST-APPLICATION

As stated, the problem of row migration can be spread across various blocks. A specific structure can be created to limit the impact to reference just two blocks to hold migrated rows. A particular original and final block repository is stored in the Migration_mapper structure if the migration is to be made. It can be associated with any table and ensures that the migration is relevant just for two blocks and cannot be spread wider.

The solution is based on identifying the data migration on the database block. Instead of using the direct pointer to the next block, the database access optimizer looks to the Migration_mapper and detects the final stage repository. Although the additional block needs to be loaded, the Migration_mapper module is commonly small and can be placed directly in the existing Buffer cache of the database instance. The I/O operation extension represents the disadvantage of the solution. Migration_mapper size demands cannot be reduced until the whole index set rebuild operation execution is done. Once again, all indexes for the particular table must be reconstructed to truncate the Migration_mapper functionality. Otherwise, it must exist original to serve the rest index access. The solution of the migration mapper is shown in Fig. 16. Note that there can be several levels of migration for the particular row. Each index can reference

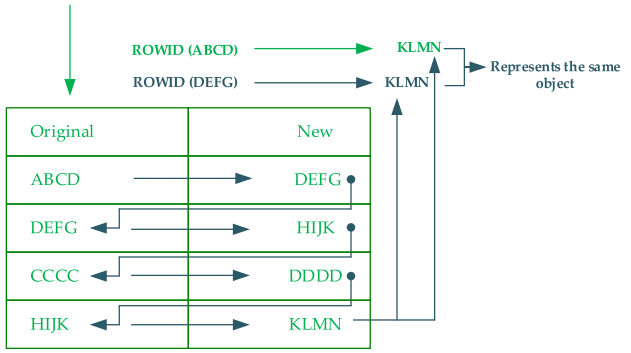

FIGURE 16. Architecture of solution 5 – Data pointer reflector.

a different root node of the Migration_mapper. Thus, the reference model cannot be directly cascaded.

E. SOLUTION 4 – STORING PATH

Another concept is covered by solution 4. Although the connection between the index and data is just one-directional, during the access using the defined index, traverse path can be temporarily stored, thus if the data migration is detected, by such definition, index pointer (ROWID) can be updated to cover the real data block, where the data are located. To implement it, a Data path reflector memory data structure is proposed. Thus, it is associated with the session. It is not shared among the instance – it would not bring any benefit, whereas the change operations on a specific row are always done in an exclusive mode – only one transaction can change a particular row at any time. Data path reflector is, therefore, associated with the transaction. After reaching the statement to terminate it (commit/abort), such a structure can be flushed. Internally, it is implemented by the pointer layer for each data update operation. If the migrated row is detected, the Session Reflector background process operator of the Data path reflector structure is notified to apply the change.

This proposed solution can, however, limit the migrated rows only partially. Provided Data reflector is interconnected only with the already used access path and does not reflect the rest indexes on the particular data table. Conversely, it applies the change only on one index element. As a result, the migrated row is fragmented. The original block must always store another block pointer to ensure data reliability and index row-level security.

F. SOLUTION 5 – DATA POINTER REFLECTOR

The last proposed solution in this category replaces the path with the direct pointers. Principles are similar to solution 4, extended by the Index Submapper background process managing the pointer infrastructure. From the architectural point of view, indexes do not hold physical database addresses (ROWIDs). Instead, logical addresses to the Data pointer reflector are used (ROWlog). Data pointer reflector is the index submapper structure transforming logical row address to the physical pointer to the database. The advantage of such an approach is based on the unicity. The migrated row creation is not reflected the individual indexes, whereas logical pointers do not evolve – they just define the source to the Data pointer reflector. So, the physical data address pointer is always stored in the database just once, irrespective of the number of defined indexes. Data pointer reflector is treated as the inline B+tree index structure based on the logical address mapping each value to one physical ROWID. Thus, if the migrated row is created, it must be applied only once in the reflector structure. Searching for the particular value can be done either by the internal B+tree index structure shaped by the reflector (solution 5a) or the traverse path (defined in solution 4) can be used, forming solution 5b. The traverse path is stored in the session-specific area and valid during the particular transaction.

Solution 5b provides significantly better performance, whereas the traverse path definition is the easier and fastest approach to reference the index. In this case, the traverse path last node extracts logical row addresses to update the physical ROWID in the reflector. As a result, migrated row is just the subelement of the processing inside the transaction. Hence, there can be no migrated rows for the managed data after committing.

Migrated row constraint removal is done no later than at the end of the transaction. Thus, it can be projected immediately after the data change itself or shifted to the end of the transaction, where multiple migrations can be applied in a common block.

The logical model of the Data pointer reflector is shown in Fig. 17. The Index Submapper background process manages the introduced memory structure. Physical ROWIDs are not used in a direct index manner. Instead, logical ROWlog pointers are used to reference the Data pointer reflector. Physical data addresses are then located just in the introduced memory structure. Migration can then be easily detected, whereas just

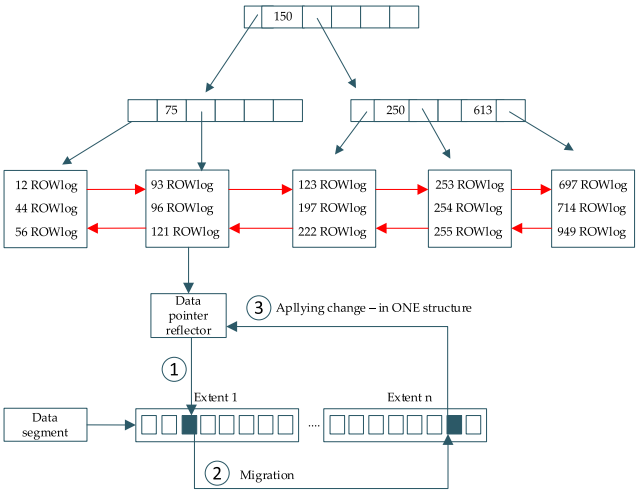

FIGURE 17. Architecture of solution 5 – Data pointer reflector.

one reference needs to be updated, the whole index set can remain original. Each index is routed to the same Data pointer reflector associated with the particular table.

G. PRIORITY MANAGEMENT

The main property of the B+tree index as a default approach is the balancing, which on the one hand, ensures performance, whereas the path from the root to any leaf is always the same. On the other hand, the individual data change operations must ensure balancing by adding additional demands. B+trees do not degrade over time and ensure efficiency with the data growth by splitting and merging blocks. Naturally, with the increase of the data amount, the index becomes more and more complex. In principle, data do not need to be accessed in an even distribution. When dealing with the time elements, current valid data forming the conventional image are most often obtained. Over time, historical data lose their meaning and are queried less and less. Index approaches do not reflect such environment characteristics and ensure the same processing priority. The research emphasis of this paper also deals with the aspect of data priority inside the index. In that case, there is no strict key balancing. Instead, the priority strategy is used to ensure that the most often used data can be obtained even significantly better.

For each data row of the registered table, the index is extended by evaluating priority, calculated by the frequency of access. It is handled in the index granularity, whereas the tuple itself is composed of attributes with various frequency, precision, and durability of the update and Select statement. Notice that each index node is extended by the Access Ticker attribute covering the amount of access. To secure the solution, each value is associated not only with the data node but encapsulates the used index access method, as well. Finally, these data modules are temporal oriented, reflecting the usage over time. Thanks to that, index balancing can be done dynamically, e.g., at the end of the month, complex analytics and reporting can be done based on monthly data granularity, so the index can be rebalanced to serve the process and focus on those covered data. Furthermore, the whole process can be done dynamically using the estimated activity list.

Evaluating the impact of the priority just on the index definition does not bring significant benefit. Reflecting on the following strategy managing the various amounts of index-covered data, it is evident that it is rather broad than deep. In section 3, there is Table 1 showing the index depth for the defined amount of data. It is calculated by the BLEVEL parameter of the {user | all | dba}_indexes data dictionary. It does not cover the root node of the index; therefore, the actual traverse path length is one greater than the dictionary obtained value.

On the other hand, the relevant block-level data can be pre-processed and preloaded into the memory based on priority. The ROWID can reference the block already present in the memory. Therefore, the I/O operation can be shifted to the less demanding processing unit period. Access Ticker consists of the total number of accesses in a compressed mode

user_index_access_ticker		
table_name	Varchar2(30)	NN (PK)
index_name	Varchar2(30)	NN (PK)
object_id	Raw(10)	NN (PK)
access_time	Date	NN (PK)
index_type	Char(1)	NN
access_method	Varchar2(30)	NN
leaf_node	Number	NN
precision_factor	Number	NN

FIGURE 18. Access Ticker data dictionary structure.

as a direct part of the index. A more detailed approach can be accessed by introducing a data dictionary view secured by the index monitoring process and used by the optimizer as part of the statistics. It consists of the following structure (Fig. 18):

- table_name – referenced table,
- index_name – the name of the used index,
- object_id – identifier (primary key) of the accessed object for the particular table,
- index_type – a type of the index – B+tree, Bitmap, Hash,
- access_method – used access method (index unique scan, range scan, etc. + methods for table joining),
- leaf_node – a reference to the accessed leaf node of the index consisting of the data address,
- access_time – date pointer of the statement execution,
- precision_factor – specifies the rebalancing precision factor of the index (used for the index optimization complexity calculation).

A usable index is always balanced and performance-optimized by applying the changes directly to the index inside the transaction. It is ensured that the transaction itself can be approved just after the data and index management. Thus, index access is trusted. Any data portion reflecting the valid data row is part of the index. As a result, if the optimizer uses the index access path, data amount mostly related to the block number is strongly limited. Therefore, the data retrieval process can significantly benefit from using the index.

On the other hand, other operations manipulating data must apply all changes to the whole index set. If the index set is strong, modifying all relevant indexes can rapidly extend the inner transaction's processing time. Three operations can be identified for each index when evaluating index management processing during the change. Firstly, it is necessary to extract data to be indexed. Then, in the second phase, detailed data are routed to the index forcing it to allocate a new index element. Such activity is placed to the traverse path, and a new node is assigned to the leaf layer. Finally, assuming that the B+tree index is used, an index balancing operation must be done.

In principle, each data row can require balancing, which can be done separately for each row (default approach), or balancing operation can be done one time on the

transaction granularity level (append hint). One way or another, index balancing restructuralizes the index by using locks to ensure that the process can be done securely and totally. Other transactions and retrieval operations must hang to ensure correctness. If the data stream is high, a significant part of the transaction management is directly related to the index balancing. One of our research projects emphasizes lowering the demands, shortening the transaction itself, and extracting the index management from the main transaction. On the other side, it is still inevitable to ensure the index's suitability, correctness, and reliability; otherwise, the index would not serve the data location. Fig. 19 shows the current data flow.

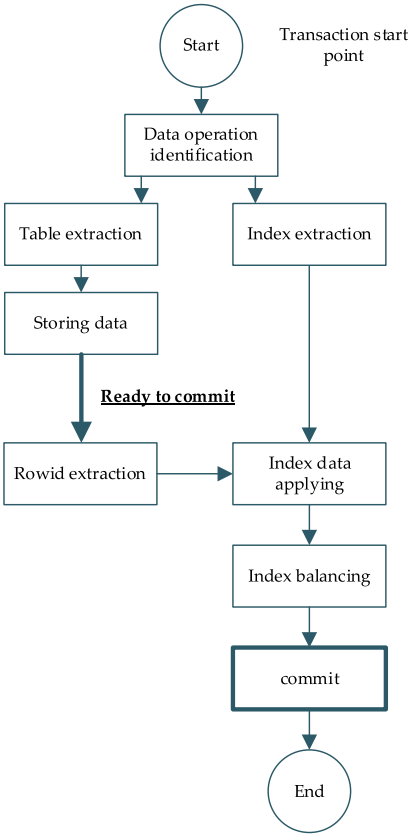

FIGURE 19. Existing data flow using the index balancing.

Our proposed solution introduces the post-transaction layer associated with the index. Data are not indexed directly inside the main transaction. Such activity is divided into two parts. A list of changes is extracted from the transaction logs during the data processing by notifying the introduced background process - Index Applier. In general, data are not directly covered by the index. Just the specific flat data structure is used for each index – Data Operator Module. If the data change vector information is placed there, the transaction can be approved and successfully ended. Thanks to that, the transaction processing time is lowered, no index balancing is present there. If there is any data inside the Data Operator Module, the Index Balancer background process becomes active. It is a master process responsible for applying changes to the relevant index, followed by balancing. It is, however, done separately from the main transaction. If the change is implemented and index balanced, the definition is removed from the associated module. As stated, an Index balancer is a master responsible process created on demand for each index. It is present during the whole period of index validity as a supervisor. The instance also has various worker processes – Index Balancer Worker(n), where "n" represents its serial number. However, these workers are not associated with the specific master process, they are just shared across the instance, and each master can put the activity for them. Fig. 20 shows the architecture and data flow of the proposed solution.

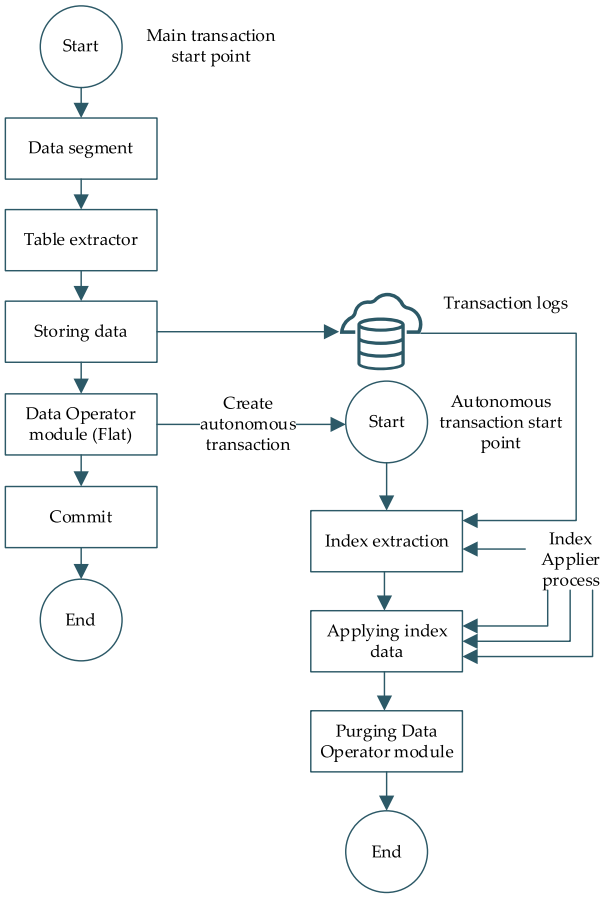

FIGURE 20. Proposed data flow using the index balancing out of the main transaction.

The proposed architecture of the post-indexing can have various benefits. As described, the main transaction can be ended sooner, whereas no index balancing is present. Balancing is done separately, operated by added processes. In contrast to existing solutions, where balancing is done on a row or statement layer, the defined solution uses transaction granularity. The balancing can be evenly distributed and cover multiple transactions in one operation. Thanks to that, the balancing strategy can be optimized and globally shortened.

More operations are applied together in one process, so the amount of balancing operations is also lowered. When dealing with data retrieval, index access can generally be used, but it must be extended by scanning the Data operator module, which is done in parallel.

H. ARCHITECTURE ENHANCEMENTS

Currently, the most often used type is B+tree based on index key balancing. Mathematical operations are applied during the index traversing to ensure the correct access path. Undefined values modeled by the NULL representation cannot be mathematically positioned and compared. As a result, undefined values are not part of the index. In [20], we have proposed several enhancements to ensure NULL value coverage. The first categorical solution is based on storing undefined values in a flat structure associated either by the left or rightmost part, or undefined tuple addresses are located in a separate structure directly interconnected to the root index element. The second approach [28] uses categories focusing on the origin of undefinition that is supervised by the transaction reliability. In that case, individual pointers are split into type buckets based on the registered modifiers. Thus, the structure is part of the index segment physically stored in the database.

In this paper, we use another perspective reflecting the memory. Instead of using the index as a physical database segment, the proposed solution locates the column store in the memory (Fig. 21). It treats individual attributes separately, even for composite indexes. Attribute column store structure is located in the memory in the B+tree shape, with the extension module for NULL value management stored separately. The background processes and memory structures represent the database instance. The memory Buffer cache is the main repository for the data blocks and indexes to be used during the data retrieval process. Introduced Memory indexer is a column repository that buckets undefined data to the categories in the NULLs management buckets. The whole structure is operated by the Memory registration process responsible for creating and maintaining the Memory indexer memory module.

VI. PERFORMANCE EVALUATION STUDY

For the performance evaluation study, the Oracle Cloud environment located in the Frankfurt data region has been used, covered by the Oracle Database 21c Enterprise Edition Release 21.0.0.0.0 – Production version 21.2.0.0.0. The whole database storage capacity was 20 GB. The database holds the spatio-temporal data dealing with the flight data by identifying the airplane objects by their flight parameters – affiliation to the airspace (entry and exit time), departure and estimated arrival time points, positional data, speed, etc. related to the defined time points. For the study, the relational model consists of 100 attributes using the following categorization:

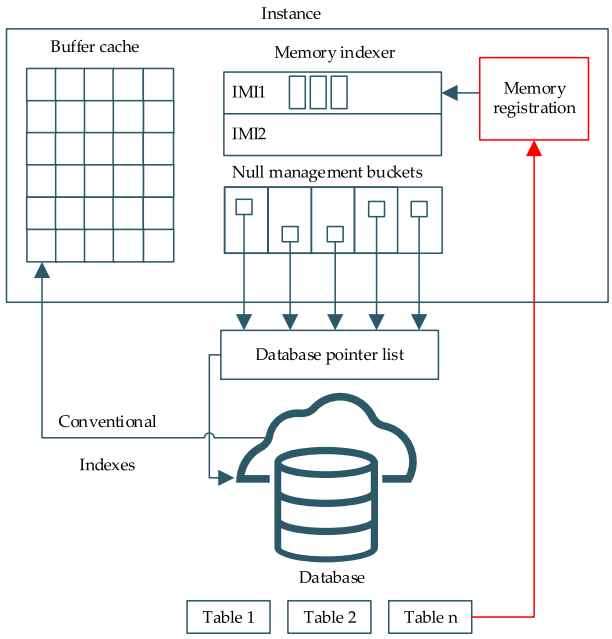

FIGURE 21. Data - Airspace assignment.

- 20 static attributes, which do not change over time (airplane and airport characteristics),
- 20 temporal attributes, which were updated synchronously by the defined frequency rate,
- 30 attributes, which were updated anytime randomly with the defined precision (if the change was not changed significantly – approximately 10% of updates, original data values remain valid),
- 30 attributes, for which the synchronization was detected and evaluated by the ML and AI techniques dynamically on the fly to minimize size demands. The group-level data management and monitoring analysis can be found in [33].

Fig. 22 shows the data example of the flight area - airspace (FIR) assignment. There is an object identifier (ECTRL ID) – airplane license plate, sequence number related to the specific flight ordering the data. AUA value identifies the flight space assignment, which can evolve over time, as well. The assignment is time-limited by the ENTRY TIME and EXIT TIME. For the processing, also the weather conditions are taken into emphasis. Planned and real routes are dynamically evaluated, highlighting flight efficiency.

The evaluated data were partitioned across the quarters for four years in total. The total data amount was 18 777 216, forming 9.5 GB of data. The rest part (10 GB) was used for tracking flight points (Fig. 23), namely flight identifier (ECTRL ID), SEQUENCE NUMBER for ordering, time point (TIME OVER), GPS position (LATITUDE, LONGITUDE), and FLIGHT LEVEL. Finally, 0.5 GB of storage was used for indexes and structural extensions.

ECTRL ID	Sequence Number	AUA ID	Entry Time	Exit Time
184408024	1	EGGXOCA	1.3.2015 5:54	1.3.2015 6:51
184408024	2	EISNCTA	1.3.2015 6:51	1.3.2015 7:17
184408024	3	EGTTCTA	1.3.2015 7:17	1.3.2015 8:00
184408024	4	ATC_UNK	1.3.2015 8:00	1.3.2015 8:00
184408024	5	EHAACTA	1.3.2015 8:00	1.3.2015 8:08
184408024	6	EHAMTMA	1.3.2015 8:08	1.3.2015 8:13
184408024	7	EHAMCTR	1.3.2015 8:13	1.3.2015 8:19

FIGURE 22. Data - Airspace assignment.

ECTRL ID	Sequence	Time Over	Flight Level	Latitude	Longitude
184408024	0	1.3.2015 1:00	0	41.98	-87.905
184408024	1	1.3.2015 1:25	0	41.98	-87.905
184408024	2	1.3.2015 1:47	330	42.06361	-84.32945
184408024	3	1.3.2015 2:05	330	42.03611	-80.75361
184408024	4	1.3.2015 2:23	330	41.89722	-77.17806
184408024	5	1.3.2015 2:41	330	41.64639	-73.60222
184408024	6	1.3.2015 3:00	330	41.28167	-70.02667
184408024	7	1.3.2015 3:15	330	41.11667	-67
184408024	8	1.3.2015 3:31	330	41.61445	-63.5

FIGURE 23. Data – Flight points.

External data files and backups were located separately in the Object storage. After the processing, if the data need to be removed from the main system, backup is created, and provided files are stored in the Cloud Object storage.

The first evaluation criterion was migrated row management. Whereas the storage capacity is limited, flight points were located in the database in a time-limited manner using the following principle:

- Current quarter assignments (flight points) are always in the system.
- Direct predecessor quarter was always present in the system.
- Older data were removed from the system dynamically using the month granularity (the strength of air traffic is most pronounced in the summer months).

Whereas the data were time delimited and supervised by the sequential object assignment, significant data migration can be present. Based on the evaluation of the flight data, 27.14% of data migration was detected on the object granularity 22.57% for the individual row granularity, reflected by the positional update of the flight point tracking.

Developed indexes were based on the following principles to ensure reliability and performance to detect the anomalies:

- flight monitoring over time,
- whole airspace monitoring over the time,
- flight corridor monitoring over the peaks,
- flight level management for the airspace, flight corridor,
- airport surroundings monitoring,
- incorrect data detection showing unrealistic changes during the time frame,
- accidents detection and risk evaluation.

Thus, seven indexes were developed, pointing to the spatio-temporal sphere – one for each above task. All of them were B+tree oriented.

In section 5, multiple solutions were presented and discussed to limit the migrated row impact. For evaluation, three aspects were processed in terms of performance:

- change database storage requirements (size demands),
- performance of the data retrieval:
  - getting relevant data of the airspace assignment (Slovakia region) during one day (24 hours) for the workday and weekend separately,
  - monitoring airplane during the whole flight (flight time in the range of one to two hours),
  - monitoring airport during one day.
- data loading and change process.

A. EVALUATION PERSPECTIVE 1 – MIGRATED ROW LIMITATION

The amount and structure of data change significantly over time. Historical images are removed or moved to archive repositories, respectively. The states evolve, which causes either the execution of the Insert command to add a new tuple state or the Update command to change the existing row dynamically, based on the representation and internal association. A typical element of data processing from various systems is the requirement to increase the processed data's accuracy and update existing states by an additional precision range. That requirement forces the system to allocate more space for individual attribute values and whole states, resulting in the necessity to allocate new blocks.

In contrast, the original space cannot fit the updated row. Thus, migrated row is to be created by decreasing the performance of the index access, whereas particular data address pointers (ROWIDs) are not precise. In this paper, several enhancements have been discussed to limit migrated rows and thus do not degrade the system's performance with a change in data - an increase in the size of the original record. The performance evaluation references the no-migration management model in a fully indexed environment. During the analysis, seven models were created and performance compared.

Namely, SOL_MIG_1 uses the data block structure extension covering the list of pointers to the index, located in the block perspective. SOL_MIG_2 uses separate address fields located in the index layer. There are two enhancements of a particular approach, SOL_MIG_2a uses inline pointer location, whereas SOL_MIG_2b uses dynamic allocation using overflow blocks. These three solutions reference pointer list on various locations and perspectives. From the size point of view, the reference model requires 4 096 MB, SOL_MIG_1 requires an additional 256 MB. The pointers extend each block to the indexes, which reference it. When moving the processing to the index layer, several references can be duplicated by increasing the demands using 384 MB for SOL_MIG_2a and 448 MB for SOL_MIG_2b. In total, additional size demands are 6.250% for SOL_MIG_1 and

approximately 10% for SOL_MIG_2 variants. The worst solution provides SOL_MIG_3, which requires a 12.500% increase in storage demands.

It is caused by the composition of the migration path using a hierarchical query. Although the data migration optimization in such a structure can be done, it requires additional system sources, but in dynamic systems, where the huge stream of updates is present, particular balancing is not relevant. Mainly, it requires too much time, and consecutively, after the processing, the structure is not balanced due to many updates in the meantime. Data path reflector is a memory structure associated with the session by flushing the data regarding the transaction. It requires an extra 384 MB. The best solutions in size demands are represented by SOL_MIG_5 variants using logical ROWlogs instead of physical representation using ROWIDs. In that case, migrated row is visible only in the Data pointer reflector and is stated only once, irrespective of the number of impacted indexes. For B+tree structure managing logical pointers, additional demands are 224 MB (SOL_MIG_5a). If the traverse path is used (SOL_MIG_5b), extra storage demands are 240 MB.

The second evaluation stream dealing with the data row migration is associated with the data retrieval. Three categories are covered – airspace assignment data, monitoring airplanes during the whole flight, and monitoring the airport. The most significant difference corresponds with the data amount to be returned and associated time-consuming.

The reference model with no migration detection requires loading multiple blocks in the chain if the original data are moved to another block. In principle, it can be done multiple times, lowering the performance, where many blocks must be loaded to obtain relevant data row. In our case, if no data migration is present, total processing time demands are 4.194 seconds for airspace assignment, 0.741 seconds for airplane monitoring, and 1.805 seconds for airport monitoring. In the evaluation study, 20% of rows were migrated physically. The index itself cannot identify it. Thus, additional demands range from 22.415% to 23.142% if no migration management is present. As evident, there is no significant difference between individual queries (less than 1%). The list of pointers reduces the processing costs from 15.803% to 16.774%. There is no overflow. Reduction is on the block level. Thus, it is still necessary to locate the original block, although the reference can be limited just to two blocks. Migration detection and management of the SOL_MIG_2 variants benefit from the index level management. The data migration is identified directly during the index processing, and no additional segments are located.

Namely, additional processing time costs range from 12.220% to 12.632% for the inline index management. Migration mapper covered by the SOL_MIG_3 does not bring significant improvement compared to already stated solutions. Therefore, the emphasis is done on the storing path in SOL_MIG_4. The traverse path for one index is temporarily stored in memory to apply the migration immediately. Total additional costs range from 8.874% to 9.602%. The limitation is associated just with one index to be covered.

On the other hand, size demands are lowered, as well. Finally, SOL_MIG_5 variants cover data pointer reflector either in the B+tree structure (SOL_MIG_5a) or traverse path can be used (SOL_MIG_5b). In both cases, processing time demands are rapidly decreased by raising only 8.140% for B+tree and 7.222% for traverse paths.

Concluding the performance analysis of the data migration, it can be stated that Data pointer reflector solutions provide the best solution and representation in both evaluated categories – size and data retrieval process.

The final evaluation stream is related to the data update operations themselves. Whereas additional structures are to be added, it is necessary to evaluate the impact on the processing. Whereas migrated rows need to be limited, it is then necessary to reconstruct the access path, mostly applied for the whole index set. All proposed solutions bring additional processing time demands. Namely, structural extensions for the block require 3.512%, which is related to the block reconstruction necessity (SOL_MIG_1). However, the whole demands are really low compared to the total percentage of the data migration (20%), which needs to be maintained. Index extension demands are increased to the value 4.173% (SOL_MIG_2a) or 4.997% (SOL_MIG_2b) caused by the index balancing necessity. Migration mapper (SOL_MIG_3) is the worst solution in the perspective of the update operation, whereas the whole access path must be composed dynamically at any time. Pointing to the solution SOL_MIG_4, the system requires only 2.102% of additional processing time, whereas only one index is treated to limit migrations inside. However, such a solution is not so robust in terms of data retrieval (an additional 10%). The best performance of the Update operations is provided by the SOL_MIG_5a (B+tree internal structure) and SOL_MIG_5b (internally operated by the traverse path). It reaches approximately 3% of additional processing time.

The complete results are mapped in the Table 3. Size, data retrieval process, and update operations are evaluated. Values are expressed in megabytes (MB) for the size, and second precision is referenced for the processing time. All values are also covered in the percentages to focus on the additional demands.

Concluding this study criterion, based on the overall performance evaluation, the best solution provides a Data reflector solution, by which the migration can be completely removed. Although it has 7% of additional processing time demands for the data retrieval operation, compared to the reference model, it is lowered up to 15% in case of using 20% of the rows, which are initially migrated. If the number of migrations rises, the ratio between the reference model and proposed SOL_MIG_5 is more significant. Namely, Table 4 shows the results comparing MIG_SOL_5b related to the reference model. It points to the migration percentage frame and related additional demands for both models. It is evident that the proposed solution is reliable and can also

TABLE 3. Results - extension requirement for data migration management.

Performance results - additional costs of the migration management			Absolute values for referential model	Extension requirements: Reference model No migration management	SOL_MIG 1	SOL_MIG 2a	SOL_MIG 2b	SOL_MIG 3	SOL_MIG 4	SOL_MIG 5a	SOL_MIG 5b
Size		MB	**4096**	0	256	384	448	384	512	224	240
		%		0	6,250	9,375	10,938	12,500	9,375	5,469	5,859
Select	getting relevant data of the airspace assignment	seconds	**4,194**	0,971	0,704	0,530	0,567	0,403	0,508	0,353	0,311
		%		23,142	16,774	12,632	13,521	12,123	9,602	8,410	7,418
	monitoring airplane during the whole flight	seconds	**0,741**	0,166	0,117	0,091	0,096	0,086	0,066	0,060	0,054
		%		22,415	15,803	12,220	12,914	11,647	8,874	8,140	7,222
	monitoring airport during one day	seconds	**1,085**	0,247	0,180	0,133	0,143	0,129	0,101	0,089	0,081
		%		22,762	16,551	12,300	13,177	11,903	9,323	8,209	7,427
Tuple Update		seconds	**14,452**	0,000	0,508	0,603	0,722	0,795	0,304	0,451	0,445
		%		0,000	3,512	4,173	4,997	5,504	2,102	3,119	3,078

TABLE 4. Results - Impact of data migration percentage.

Data retrieval - Extension requirements			
migrated row ratio	Reference model No migration management	SOL_MIG 5b	Difference
	%	%	%
20	22,773	7,356	15,417
30	34,759	8,647	26,112
40	48,756	9,129	39,627
50	60,683	11,237	49,446
60	76,783	12,975	63,808
70	87,112	14,012	73,100
80	99,742	15,800	83,942
90	112,401	17,468	94,933
100	137,820	20,771	117,049

ensure the performance in the degraded physical architecture. For declarative purposes, values are expressed in percentage.

Fig. 24 shows the results in a graphical form.

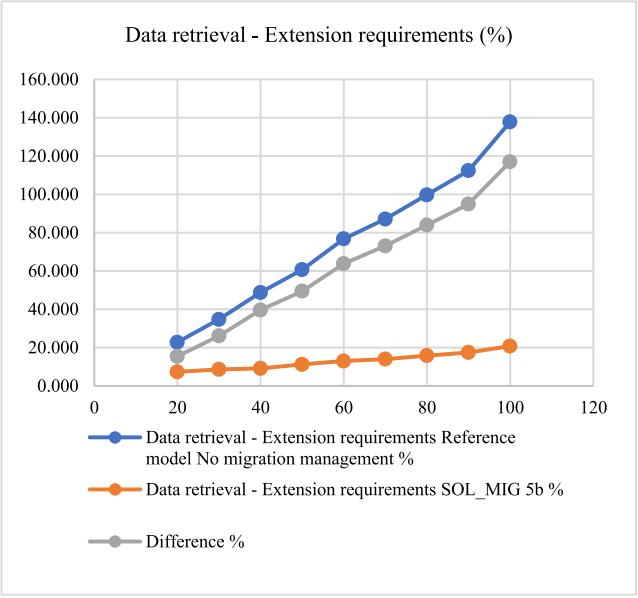

FIGURE 24. Results - Impact of SOL_MIG_5b on the performance.

B. EVALUATION PERSPECTIVE 2 – PRIORITY MANAGEMENT

In the real-time control and safety systems, it is inevitable to minimize the time consumption to obtain relevant data. Section 5 deals with the own solution by covering priority in the B+tree structure extension. This evaluation discusses the impact and benefits of such a model by the following aspects:

- change database storage requirements (size demands),
- performance of the data retrieval:
  - getting relevant data of the airspace assignment (Slovakia region) during one day (24 hours) for the workday and weekend separately,
  - monitoring airplane during the whole flight (flight time in the range of one to two hours),
  - monitoring airport during one day),
- rebalancing time demands and total costs.

The focus is on the data retrieval process during the performance analysis and evaluation, which can lower the demands if the relevant data are directly accessible via index without individual node loading necessity. In this case, we use three indexes covering the above-listed data retrieval processes. As evident, managing priority brings additional size demands in two sources:

- Index node extension dealing with the total number of touches (SOL_PRIORITY_1). This solution provides only a basic overview of the index node usage. There is no definition of the access method nor the index and table structure optimization level (block structure fragmentation). However, the whole data are covered directly inside the node allowing the system to load all data together, forming the main benefit. Although the size demands are not extended significantly, a relatively

robust and efficient solution can be achieved in terms of priority detection.

- Data dictionary extension dealing with the more comprehensive statistics, like used access method, the total amount of data covered by the query, executed operation (Insert, Update, Delete or Select statement), estimated and real costs, etc. (SOL_PRIORITY_2). Individual access methods can have various significance levels, namely, unique and range scans focus on retrieving data using the optimal index. Indirect data access is performed by the full index or fast full index scan methods, by which the whole index is searched. In that case, the touch element is not associated with the individual data nodes, but the entire index is referenced. This solution requires a bigger storage extension by referencing table, index, individual index nodes, and methods. All the data are temporal, losing relevance over time to ensure that the most up-to-date data are preferred.

Table 5 shows the size demands in MB. The first solution does not deal with priority management, holding the original index tree structure (SOL_PRIORITY_REF). SOL_PRIORITY_1 uses data block extension for each leaf element and requires 6 144 MB, reflecting the difference of 2 048 MB in total. Dealing with the extent granularity consisting of eight blocks of 8kB, additional 32 extents are necessary to be used. The total depth of the index was 3. When dealing with the SOL_PRIORITY_2, additional size demands are identified for the data dictionary holding the touches and references, emphasizing the used access method. Compared to the original solution with no priority management, the required increase in disk space is 3 072 MB. These data are provided automatically during the statistics refresh or by individual operation accessing the data. Comparing both developed solutions, SOL_PRIORITY_2 uses 1 024 MB of additional space. However, it reflects the access method and the weight, significance, and time reference. As stated, the system should focus on the most up-to-date references and data locations.

TABLE 5. Priority management – size demands.

Name	SOL PRIORITY_REF	SOL PRIORITY_1	SOL PRIORITY_2
Type	No priority management	Internal priority management (leaf index nodes)	Extended data dictionary statistics
Size demands (MB)	4096	6144	7168

Table 5 shows the total size demands for the indexes. It is, however, necessary to evaluate the positive impact on the data retrieval operation performance. Namely, we cover airspace assignment flight and airport monitoring. The total processing demands are expressed in Table 6 using the second precision. As can be seen from the results, there is just a slight performance benefit for the whole airspace monitoring.

TABLE 6. Priority management – time processing.

Name	SOL PRIORITY_REF	SOL PRIORITY1	SOL PRIORITY2
Type	No priority management	Internal priority management (leaf index nodes)	Extended data dictionary statistics
getting relevant data of the airspace assignment (seconds)	4.194	4.142	4.073
monitoring airplane during the whole flight (seconds)	0.743	0.741	0.739
monitoring airport during one day (seconds)	1.823	1.805	1.784

Without any priority management, the total time demands are 4.194 seconds. By using proposed optimization, demands are lowered using 1,26% for SOL_PRIORITY_1 and 2,67% for SOL_PRIORITY_2. As visible, although there are benefits in terms of processing time, the additional demands in size do not make them up. Using flight monitoring, even less significant difference expressed in percentage is present – no more than 0.6% at the microsecond level.

Similarly, monitoring the airport during one day does not benefit from using priority. There is just a little difference, up to 2.19%, for comprehensive data dictionary statistics management (SOL_PRIORITY2). The main reason is the index depth. If the table holds no more than 1 billion rows, the index depth is 3 or 4, respectively. As a result, massive rebalancing does not bring additional power. When dealing with more data amounts, only an insignificant change is present. Note that if GB or TB od data are stored in one table, commonly, data are partitioned using a local index set for each sub-structure, shifting the solution to the already described model.

Table 7 shows the demands on the rebalancing in time representation (second precision).

TABLE 7. Priority management – rebuilding.

Name	SOL PRIORITY_REF	SOL PRIORITY_1	SOL PRIORITY_2
Type	No priority management	Internal priority management (leaf index nodes)	Extended data dictionary statistics
Rebuilding time demands (seconds)	109.254	131.925	128.854

Balancing based on the priority brings additional demands. If the touches of individual leaf index nodes directly inside are stored internally in the index, before rebuilding, such data

must be extracted and temporarily stored in the database. Moreover, index size is extended by storing access frequency for each row. As a result, an additional 22.671 seconds are required for the specified environment. The main advantage of the extended storage used by the solution SOL_PRIORITY_2 is based on the fact that the index structure remains the same in terms of structure. Therefore, there are no additional blocks required.

On the other hand, it is necessary to calculate the ratio and priority for each node to be applied. It brings 19.6 seconds for such activity. Fig. 25 shows the results in the graphical form.

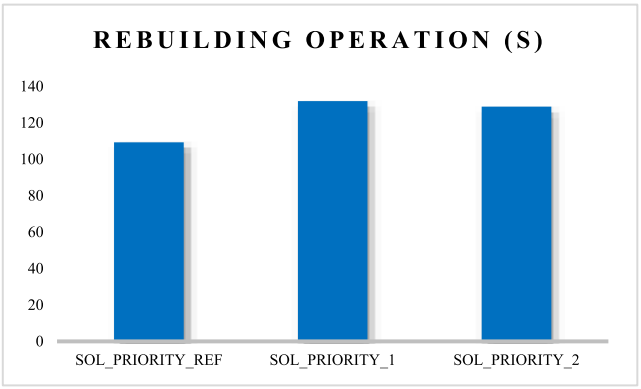

FIGURE 25. Rebuilding operation demands – priority management results (seconds).

The second criterion in this part is related to the balancing operation based on priority. Three environments are to be used – with no priority management, internal and external data source. Table 8 shows the results for 100 000 balancing operations based on the same environment.

TABLE 8. Rebalancing time – priority management.

Name	SOL PRIORITY_REF	SOL PRIORITY_1	SOL PRIORITY_2
Type	No priority management	Internal priority management (leaf index nodes)	Extended data dictionary statistics
Rebalancing time demands for 100 000 operations (seconds)	17.489	21.176	20.813

In this experiment, we have dealt with priority management covering the whole data management spectrum. Firstly, we covered the storage demands for the processing. Secondly, the performance of the data retrieval has been monitored using proposed schemes. Based on the study, it can be concluded that the priority management, even for the critical data, is not relevant and does not provide sufficient benefits. It is clear that the processing time duration was lowered just slightly, but on the other hand, significant storage capacity demand extensions were applied. Finally, rebuild and balancing operations were evaluated to handle priorities. Index rebalancing reflecting the priority requires additional 3.687 seconds for internal management and 3.324 seconds. Comparing internal and external management, there is some small increase. Based on the further analysis, it is caused by the necessity to store the index leaf touches, which forces the structure to extend block size and the need to store more data consecutively.

C. EVALUATION PERSPECTIVE 3 – INDEXING OUTSIDE THE MAIN TRANSACTION

Sensor systems provide big data flow to be evaluated, processed, and consecutively stored. Ensuring proper data management in industry, control systems, medicine, or traffic management systems is inevitable. Any delay can be disastrous and catastrophic. Consequently, transactions should be approved immediately after checking integrity rules to ensure the reliability of the data for the security layer and the whole control systems and decision-making. However, the second aspect is associated with indexing, guaranteeing a balanced structure inside the main transaction. This balancing operation is commonly part of the transaction. This evaluation perspective analyzes the impact of indexing directly in the core transaction compared to autonomous management separately. The external balancing process is primarily intended for systems, covering multiple indexes. Individual change operations are heavily present in the system to optimize them to minimize processing time and costs. Thus, the goal is to effectively cover changes over data over time on the one hand, but the process of obtaining and accessing the data tuples must be efficient as well. It is primarily ensured by the indexes so that any change above them is applied autonomously through the proposed Balancer background processes.

For the evaluation, three approaches are used. The first model (SOL_NO_IND_REF) is a reference model that does not manage indexes. Thus, there are no additional demands regarding indexing during the Insert, Update or Delete statement. The second model (SOL_IND_COMMON) uses the common principle of transaction coverage inside the main transaction. Like the discussed environment, three indexes cover airspace assignments during one day, airplane route monitoring and airport space monitoring. The last solution deals with the proposed indexing strategy (SOL_IND_EXTERNAL). Performance study is done for the data retrieval and inserting a new state to the database.

Table 9 shows the results. Without using an index, sequential scanning is inevitable to be executed. Such a model is used as a 100% reference (SOL_NO_IND). Index set management inside the direct transaction requires an additional 3.11 seconds, which expresses the coverage by the index and the balancing operation. For this model, it is done for each Insert operation separately. Using APPEND hint (index is balanced after the change operation just once for all data), the total processing time elapses 16.872, reaching just 2.42 seconds for the index management. Compared to the SOL_IND_COMMON, the defined hint lowers the demand using 0.69 seconds, reflecting the 28.51% improvement.

TABLE 9. Data retrieval process.

		SOL NO_IND REF	SOL IND COMMON	SOL IND EXTERNAL
Insert	Insert – 100 000 rows	14.452	17.562	14.789
Select	getting relevant data of the airspace assignment (seconds)	17.787	4.194	4.623
	monitoring airplane during the whole flight (seconds)	6.741	0.743	0.883
	monitoring airport during one day (seconds)	9.102	1.823	2.073

The difference between no index (SOL_NO_IND_REF) and external index management (SOL_IND_EXTERNAL) is made by the notification necessity to ensure consecutive balancing in an autonomous transaction. Such property causes a little overhead – 0.337 seconds (2.28%). Thus massive indexing can be done with minimal impact on the data input processing.

Reflecting the data retrieval performance using the defined criteria, additional processing time demands range from 12 to 15% for the sequential processing. By shifting the solution to the parallel environment – one process deals with the index itself, and the second worker process operates the unprocessed nodes of the index in a flat layer, additional processing demands can be lowered to 5-6%.

D. EVALUATION PERSPECTIVE 4 – DATA COMPLEXITY AND RELIABILITY

The last evaluation strategy points to the data relevance. Whereas the communication channel cannot be done by 100% trusted interconnection via cable without any error, detecting either the delays or improper data reflecting the failure is important. Data in these terms can be even broken (out of range, non-relevant, or not complying with the shapes and patterns) or proposed in a non-suitable time reflection caused by the delays at various levels. Furthermore, whereas the interconnection is usually done via the wireless network with the possibility of a loss of connectivity or signal quality, it is also necessary to identify such situations to ensure reliability. Thus, three cases covering the data consistency can be identified:

- Data were not obtained at all. In that case, if the synchronous process provides the value, then the NULL value is commonly used for the representation.
- Provided data are not relevant, meaning that the reliability issue is identified on the transaction consistency and integrity. The state is then stored either by NULL values or by storing original data with the reliability mark.
- Data are delayed. Similarly, the processing depends on the status and data frequency. Using synchronization, the NULL value is used. Otherwise, the temporal model is extended by the data request time and the actual image acquisition time.

TABLE 10. Undefined value management - results.

Model	SOL NO_NULL COVERAGE	SOL NULL FUNCTION	SOL NULL DYNAMIC	SOL NULL INDEX
Storage demands (%)	100	106.245	100	101.471
Data retrieval (%)	100	34.660	87.652	14.742

As described, NULL values form an inseparable part of the data coverage and reliability. In common B+tree indexing, such values are not part, and their identification requires sequential data block-by-block scanning resulting in various limitations:

- data blocks are typically fragmented,
- empty blocks can be present in the system (as a result of migrating data to a historical repository),
- significant data amount needs to be memory loaded and evaluated, and there is no direct pointer to the data tuple inside the block,
- specific geographic (airspace), time positions, or regional data cannot be processed directly, resulting in the whole table scanning necessity – data conditions cannot be applied by using sequential scanning,
- there is no evidence of whether the NULL value exists in the system.

NULL value management is a significant element to cover the reliability and security of the system by dealing with non-trusted or delayed data. To evaluate the proposed solution, four models have been used:

- no explicit NULL management (SOL_NO_NULL_COVERAGE) resulting in sequential scanning necessity,
- NULL value function transformation in the input stream (SOL_NULL_FUNCTION),
- dynamic transformation (SOL_NULL_DYNAMIC),
- proposed solution by index coverage (SOL_NULL_INDEX).

The obtained results are shown in Table 10. Values;; are correlated in percentage to declare the impact. The particular table consists of 10% of undefined or untrusted data tuples. By the data retrieval process, such undefined values are to be located. SOL_NO_NULL_COVERAGE is a reference model holding 100%.

Undefined value management by transforming the input stream using the function call brings additionally 6.245% of the storage demands. The transformed values are processed at the input, and thus the replaced values are physically

stored in the database and consecutively indexed. The physical representation on the logical layer uses specific database address denotation. Dynamic NULL identification during the retrieval does not require extra storage. Original NULL values are stored. Finally, SOL_NULL_INDEX covers the undefined values directly inside the index. Various techniques from the interval representation can be used; however, the physical storage demands are the same, pointing to 1.471%.

To conclude such part, functional transformation reaches the database, as well as the index by new data blocks. Direct NULL value modeling inside the index is delimited by 1.5% increased storage costs. There is no database extension necessity. The difference between SOL_NULL_INDEX and SOL_NULL_FUNCTION expresses the 4.8% increase using the database block layer.

Although an increase in the level of the data storage is identified in the processing, a considerable improvement can be obtained in the processing of the data in terms of access to them. The reference model (SOL_NO_NULL_COVERAGE) does not deal with the undefined values, reaching 100%. It requires sequential data block scanning irrespective of its usage or fragmentation. By transforming undefined values using a function call, the total requirements represented by the processing time are 34.7%. In this case, the index can obtain data block addresses to be loaded. Dynamic transformation is not so powerful, although some performance gain can be located, up to 12.3%. However, there is no storage capacity extension. Finally, the best solution was obtained by the proposed SOL_NULL_INDEX, reducing the demands to 14.7%. Specifically, 1.3% is used for the direct index management supervision (administrative instance tasks). Undefined storage management inside the index requires 13.447%. In the data retrieval process, all undefined values are obtained during the query (10% in total), the categorization tool processed 2.625% of the total costs.

In conclusion, we can unambiguously declare that the required reduction of time costs is provided by the proposed solution, which expands the capacity of the index to cover undefined states, either in the standard form or by expanding the solution by categorizing the causes of undefined value. As we can see, such categorization is not resource-demanding, but it can be significantly beneficial in error identification, which can be crucial in many systems. Performance study has been done on the flight data model management based on the sensor data processing but can be generalized to any industrial or data management sphere.

VII. CONCLUSION

Over the decades, data management structures changed. However, the area that persists to this day is relational database technology. The main advantage is the strict model definition supervised by the integrity rules and data consistency. These parts are secured by the transactions ensuring the transformation from one consistent data image to another, which will also be consistent. Any approved data are durable and accessible by the database. It is important to commit the transaction concerning the business environment and inner requirements to ensure data complexity. However, it is necessary to emphasize the internal database rules to make the transaction commit as soon as possible, minimize the time demands costs, and make the changes visible by the other systems and sessions.

However, such a requirement is limited by the developed index set for the particular table. Thus, one strong condition is to ensure fast data accessibility by the indexes. However, the other perspective is related to the process of data obtaining, evaluating, and storing new tuples in the database.

This paper deals with index management by extending the structure. The common index type is B+tree, which is always balanced, forcing the transaction manager to ensure such a process directly inside the transaction. Our proposed solution excludes such operations to separate transactions by using Data indexers. Thanks to that, the original transaction can be approved sooner, the balancing benefits from grouping multiple states to be applied at once. Furthermore, as discussed in the evaluation study, it significantly improves data manipulation. Parallel processing of the data retrieval ensures that the index and introduced memory structures are scanned simultaneously with minimal impact on the processing demands (time and costs). Thanks to the proposed architecture, data can be processed and evaluated sooner, but the index set is reliable, covering all the data with the extensional modules.

Another problem discussed in this paper is associated with reliability. Undefined values are often present in the system formed by various origins. These data are marked as non-reliable, physically represented by the NULL values. Such values cannot be part of the index, whereas NULL values cannot be mathematically compared and used for traverse paths. It results in the whole table scanning necessity, block-by-block. Our proposed solution is based on index extension to cover undefinition either at a pure level or by using the origin categorization. Therefore, undefined values can be part of the index extension modeled by the bucket arrays or sorted tree. Sequential block scanning is then replaced by index search, which provides significant performance improvement in reducing the data amount to be loaded.

Moreover, it is independent of the physical structure, and data fragmentation does not play any role. In the computational study, it is declared that such a module brings a strong decrease in processing costs, mostly caused by the efficiency of the processing and I/O operations. Note that the indexes are commonly at least partially stored in the memory. One way or another, the index set is optimized for size when changing.

Fragmentation is just one element of the physical architecture in terms of blocks. The second problem is associated with the free blocks due to moving data to archive repositories. Delayed or improperly obtained and evaluated data can be consecutively changed to declare the precision. These circumstances form the basis for creating migrated rows, where the original record after the change can no longer be processed and stored in the original block due to its size. The migrated row is represented by storing the address of the next

block in which the record is located. As the block itself does not store index references, additional costs are incurred for data access. The index leaf node obtains the address of the block (ROWID), which is read into the instance's memory to extract the record. In reality, however, the record is not there, and it is necessary to locate another block. In general, it may be required to process multiple blocks to obtain the needed record itself. This paper proposes new structures and background processes responsible for identifying migrated rows and applying changes. Several solutions were proposed, covered by the performance and limitations. The dynamic mapping structure does the most powerful solution. Physical ROWIDs are replaced by the logical addresses (ROWlogs) to the mapping structure storing physical addresses. Thus, the additional layer is introduced, by which the migration can be easily detected.

Moreover, any change is always at the level of only one record, regardless of the number and structure of the indexes. Thus, such a mapping structure can bring significant performance benefits, whereas there is just one place to place and locate migration. Although additional data structure is used, total costs and processing time benefit.

In the future development, our emphasis will focus on the parallelism and distributed environment, by which the execution plan can be different based on the cost estimation. We will analyze the impact of parallel thread amount on the index access method and synchronization process. Various spatio-temporal architectures will be treated with emphasis on data grouping. Another research stream will be related to dynamic block size management. It is based on the assumption that with the rise of the data amount, data granularity evolves as well. It is, therefore, necessary to define the corresponding base block size and structuralize the internal layer dynamically to respond to pattern changes at the block size level. As a result, each segment can be delimited by the different block size, even applied for each extent separately. However, it would also require changes at the memory level and the process of loading data into memory. We also want to focus on creating such a dynamic architecture to optimize the data storage.

REFERENCES

[1] W. S. Lima, H. L. S. Bragança, and E. J. P. Souto, "NOHAR—NOvelty discrete data stream for human activity recognition based on smartphones with inertial sensors," *Expert Syst. Appl.*, vol. 166, Art. no. 114093, Mar. 2021, doi: 10.1016/j.eswa.2020.114093.

[2] P. Stone, "Machine learning for robot locomotion: Grounded simulation learning and adaptive planner parameter learning," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2022, p. 6, doi: 10.1109/BIGDATA52589.2021.9671792.

[3] I. Mlynkova and J. Pokorny, "Similarity of XML schema fragments based on XML data statistics," in *Proc. Innov. Inf. Technol. (IIT)*, Nov. 2007, pp. 243–247, doi: 10.1109/IIT.2007.4430402.

[4] D. Jin, G. Chen, W. Hao, and L. Bin, "Whole database retrieval method of general relational database based on lucene," in *Proc. IEEE Int. Conf. Artif. Intell. Comput. Appl. (ICAICA)*, Jun. 2020, pp. 1277–1279, doi: 10.1109/ICAICA50127.2020.9182496.

[5] R. Wang, B. Salzberg, and D. Lomet, "Transaction support for log-based middleware server recovery," in *Proc. IEEE 25th Int. Conf. Data Eng.*, Mar. 2009, pp. 353–356, doi: 10.1109/ICDE.2009.45.

[6] Y. Tang, L. Chen, J. Liu, and D. Li, "Speeding up virtualized transaction logging with vTrans," in *Proc. IEEE 22nd Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2016, pp. 916–923, doi: 10.1109/ICPADS.2016.0123.

[7] J. Hoon Jang, S. M. Lee, S. D. Kim, O. Seong Gwon, E. Ko, S. M. Lee, J. Woo Shin, and S. E. Lee, "Accelerating forex trading system through transaction log compression," in *Proc. Int. SoC Design Conf. (ISOCC)*, Nov. 2014, pp. 74–75, doi: 10.1109/ISOCC.2014.7087602.

[8] Y.-L. Lo and C.-Y. Tan, "A study on multi-attribute database indexing on cloud system," in *Proc. Int. MultiConf. Eng. Comput. Scientists (IMECS)* (Lecture Notes in Engineering and Computer Science), vol. 2195. Cham, Switzerland: Springer, Mar. 2012, pp. 299–304.

[9] Q. He, F. Zhang, G. Bian, W. Zhang, D. Duan, Z. Li, and C. Chen, "Research on data routing strategy of deduplication in cloud environment," *IEEE Access*, vol. 10, pp. 9529–9542, 2022, doi: 10.1109/ACCESS.2021.3139757.

[10] O. Rolik, K. Ulianytska, M. Khmeliuk, V. Khmeliuk, and U. Kolomiiets, "Increase efficiency of relational databases using instruments of second normal form," in *Proc. IEEE 3rd Int. Conf. Adv. Trends Inf. Theory (ATIT)*, Dec. 2021, pp. 221–225, doi: 10.1109/ATIT54053.2021.9678605.

[11] G. Graefe, W. Guy, and C. Sauer, "Instant recovery with write-ahead logging: Page repair, system restart, media restore, and system failover, second edition," *Synth. Lectures Data Manage.*, vol. 8, no. 2, pp. 1–113, Apr. 2016, doi: 10.2200/S00710ED2V01Y201603DTM044.

[12] V. Ottaviani, A. Lentini, A. Grillo, S. Di Cesare, and G. F. Italiano, "Shared backup & restore: Save, recover and share personal information into closed groups of smartphones," in *Proc. 4th IFIP Int. Conf. New Technol., Mobility Secur.*, Feb. 2011, pp. 1–5, doi: 10.1109/NTMS.2011.5720655.

[13] M. A. Radaideh, "A distributed and parallel model for high-performance indexing of database content," *Int. J. Comput. Appl.*, vol. 26, no. 4, pp. 257–262, 2004, doi: 10.2316/journal.202.2004.4.202-1404.

[14] G. Arora, S. Kalra, A. Bhatia, and K. Tiwari, "PalmHashNet: Palmprint hashing network for indexing large databases to boost identification," *IEEE Access*, vol. 9, pp. 145912–145928, 2021, doi: 10.1109/ACCESS.2021.3123291.

[15] H. Chen and J. Li, "The research of embedded database hybrid indexing mechanism based on dynamic hashing," in *Proc. Int. Conf. Inf. Technol. Softw. Eng.* (Lecture Notes in Electrical Engineering), vol. 211. Berlin, Germany: Springer, 2013, pp. 691–697, doi: 10.1007/978-3-642-34522-7_74.

[16] M. S. Lew, D. P. Huijsmans, and D. Denteneer, "Optimal keys for image database indexing," in *Image Analysis and Processing*, vol. 1311. Berlin, Germany: Springer, 1997, pp. 148–155, doi: 10.1007/3-540-63508-4_117.

[17] M. Lorenzini, W. Kim, and A. Ajoudani, "An online multi-index approach to human ergonomics assessment in the workplace," *IEEE Trans. Human-Mach. Syst.*, early access, Jan. 7, 2022, doi: 10.1109/THMS.2021.3133807.

[18] H. Khatri, J. Fan, Y. Chen, and S. Kambhampati, "QPIAD: Query processing over incomplete autonomous databases," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, Apr. 2007, pp. 1430–1432, doi: 10.1109/ICDE.2007.369028.

[19] M. Varga, M. Kvassay, and M. Kvet, "Teaching course on algorithms and data structures during the coronavirus pandemic," in *Proc. 18th Int. Conf. Emerg. eLearning Technol. Appl. (ICETA)*, Nov. 2020, pp. 730–738, doi: 10.1109/ICETA51985.2020.9379156.

[20] M. Kvet and M. Kvet, "Relational pre-indexing layer supervised by the DB_index_consolidator Background Process," in *Proc. 28th Conf. Open Innov. Assoc. (FRUCT)*, Jan. 2021, pp. 222–229, doi: 10.23919/FRUCT50888.2021.9347573.

[21] D.-H.-T. That, M. Gharehdaghi, A. Rasin, and T. Malik, "LDI: Learned distribution index for column stores," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2021, pp. 376–387, doi: 10.1109/BIGDATA52589.2021.9671318.

[22] H. Huang and H. Luan, "Rethinking insertions to B^+-trees on coupled CPU-GPU architectures," in *Proc. IEEE Int. Conf. Parallel Distrib. Process. Appl., Big Data Cloud Comput., Sustain. Comput. Commun., Social Comput. Netw. (ISPA/BDCloud/SocialCom/SustainCom)*, Sep./Oct. 2021, pp. 993–1001, doi: 10.1109/ISPA-BDCLOUD-SOCIALCOM-SUSTAINCOM52081.2021.00139.

[23] S. V. Oprea, A. Bara, V. Diaconita, C. Ceaparu, and A. A. Ducman, "Big data processing for commercial buildings and assessing flexibility in the context of citizen energy communities," *IEEE Access*, vol. 9, pp. 168715–168730, 2021, doi: 10.1109/ACCESS.2021.3137352.

[24] D. Kuhn and T. Kyte, *Oracle Database Transactions and Locking Revealed*. Berkeley, CA, USA: Apress, 2021, doi: 10.1007/978-1-4842-6425-6.

[25] B. M. Abdelhafiz, "Distributed database using sharding database architecture," in *Proc. IEEE Asia–Pacific Conf. Comput. Sci. Data Eng. (CSDE)*, Dec. 2020, pp. 1–17, doi: 10.1109/CSDE50874.2020.9411547.

[26] M. Tareq, E. A. Sundararajan, A. Harwood, and A. A. Bakar, "A systematic review of density grid-based clustering for data streams," *IEEE Access*, vol. 10, pp. 579–596, 2021, doi: 10.1109/ACCESS.2021.3134704.
[27] A. E. A. Raouf, A. Abo-Alian, and N. L. Badr, "A predictive multi-tenant database migration and replication in the cloud environment," *IEEE Access*, vol. 9, pp. 152015–152031, 2021, doi: 10.1109/ACCESS.2021.3126582.
[28] M. Kvet, "Database index balancing strategy," in *Proc. 29th Conf. Open Innov. Assoc. (FRUCT)*, May 2021, pp. 214–221, doi: 10.23919/FRUCT52173.2021.9435452.
[29] M. Kratky, V. Snasel, J. Pokorny, and P. Zezula, "Efficient processing of narrow range queries in multi-dimensional data structures," in *Proc. 10th Int. Database Eng. Appl. Symp. (IDEAS)*, Dec. 2006, pp. 69–79, doi: 10.1109/IDEAS.2006.21.
[30] C.-H. Cheng, L.-Y. Wei, and T.-C. Lin, "Improving relational database quality based on adaptive learning method for estimating null value," in *Proc. 2nd Int. Conf. Innov. Comput., Inf. Control (ICICIC)*, Sep. 2007, p. 81, doi: 10.1109/ICICIC.2007.350.
[31] M. N. Islam, P. C. Shill, M. F. Mridha, D. M. S. Islam, and M. M. A. Hashem, "Generating weighted fuzzy rules for estimating null values using an evolutionary algorithm," in *Proc. Int. Conf. Elect. Comput. Eng.*, Dec. 2006, pp. 324–327, doi: 10.1109/ICECE.2006.355637.
[32] M. Kao, N. Cercone, and W.-S. Luk, "Providing quality responses with natural language interfaces: The null value problem," *IEEE Trans. Softw. Eng.*, vol. 14, no. 7, pp. 959–984, Jul. 1988, doi: 10.1109/32.42738.
[33] M. Kvet and K. Matiasko, "Efficiency of the relational database tuple access," in *Proc. IEEE 15th Int. Scientific Conf. Informat.*, Nov. 2019, pp. 231–236, doi: 10.1109/INFORMATICS47936.2019.9119325.
[34] C. Jensen, C. Christian, S. Jensen, R. Snodgrass, M. Böhlen, R. Busatto, H. Gregersen, K. Torp, A. Datta, and S. Ram, "Temporal data management," *IEEE Trans. Knowl. Data Eng.*, vol. 11, no. 1, Jan./Feb. 1999.
[35] I. B. E. Dunaieva, V. Vecherkov, V. Popovych, V. Pashtetsky, V. Terleev, A. Nikonorov, and L. Akimov, "Spatial and temporal databases for decision making and forecasting," in *Energy Management of Municipal Transportation Facilities and Transport* (Advances in Intelligent Systems and Computing), vol. 1259. Cham, Switzerland: Springer, 2019, pp. 198–205, doi: 10.1007/978-3-030-57453-6_17.
[36] M. Kvet, R. Ceresnak, and V. Salgova, "Use of machine learning for the unknown values in database transformation processes," in *Proc. Commun. Inf. Technol. (KIT)*, Oct. 2021, pp. 1–7, doi: 10.1109/KIT52904.2021.9583753.
[37] D. Kuhn, S. R. Alapati, and B. Padfield, *Expert Oracle Indexing and Access Paths*. Berkeley, CA, USA: Apress, 2016, doi: 10.1007/978-1-4842-1984-3.
[38] H. Liu, Q. Chen, N. Pan, Y. Sun, Y. An, and D. Pan, "UAV stocktaking task-planning for industrial warehouses based on the improved hybrid differential evolution algorithm," *IEEE Trans. Ind. Informat.*, vol. 18, no. 1, pp. 582–591, Jan. 2022, doi: 10.1109/TII.2021.3054172.
[39] M. Kvet, "Study of duplicate tuple management," in *Proc. IEEE Int. Conf. Syst., Man, Cybern. (SMC)*, Oct. 2021, pp. 3081–3088, doi: 10.1109/SMC52423.2021.9658726.
[40] D. Kuhn and T. Kyte, *Expert Oracle Database Architecture*. Berkeley, CA, USA: Apress, 2022, doi: 10.1007/978-1-4842-7499-6.
[41] K. Unnikrishnan and K. V. Pramod, "On implementing temporal coalescing in temporal databases implemented on top of relational database systems," in *Proc. Int. Conf. Adv. Comput., Commun. Control*, New York, NY, USA, 2009, pp. 153–156.
[42] Z. Deng, Z. Deng, Y. Wang, T. Liu, S. Dustdar, R. Ranjan, A. Zomaya, Y. Liu, and L. Wang, "Spatial-keyword skyline publish/subscribe query processing over distributed sliding window streaming data," *IEEE Trans. Comput.*, early access, Jan. 6, 2022, doi: 10.1109/TC.2022.3140884.
[43] F. Farahnakian, L. Koivunen, T. Makila, and J. Heikkonen, "Towards autonomous industrial warehouse inspection," in *Proc. 26th Int. Conf. Autom. Comput. (ICAC)*, Sep. 2021, pp. 1–6, doi: 10.23919/ICAC50006.2021.9594180.
[44] G. S. Nair, P. V. Devika, K. Jyothisree, N. Giriraj, and S. N. B. Sai, "Architecture, concept and algorithm for data analytics based zero touch waste management in smart cities," in *Proc. 2nd Int. Conf. Electron. Sustain. Commun. Syst. (ICESC)*, Aug. 2021, pp. 851–856, doi: 10.1109/ICESC51422.2021.9532723.
[45] S. Al-Azani, S. M. Sait, and K. A. Al-Utaibi, "A comprehensive literature review on children's databases for machine learning applications," *IEEE Access*, vol. 10, pp. 12262–12285, 2022, doi: 10.1109/ACCESS.2022.3146008.
[46] J. Tims, R. Gupta, and M. L. Soffa, "Data flow analysis driven dynamic data partitioning," in *4th Int. Workshop Lang., Compil., Run-Time Syst. Scalable Comput.* Berlin, Germany: Springer-Verlag, 1998, pp. 75–90.
[47] J. Wang, Z. Duan, X. Han, and D. Yang, "Efficient top/bottom-K fraction estimation in spatial databases using bounded main memory," *Tsinghua Sci. Technol.*, vol. 27, no. 2, pp. 223–234, Apr. 2022, doi: 10.26599/TST.2021.9010020.
[48] A. H. Al-Sanhani, A. Hamdan, A. B. Al-Thaher, and A. Al-Dahoud, "A comparative analysis of data fragmentation in distributed database," in *Proc. 8th Int. Conf. Inf. Technol. (ICIT)*, May 2017, 724-729, doi: 10.1109/ICITECH.2017.8079934.
[49] C. I. Ezeife and J. Zheng, "Measuring the performance of database object horizontal fragmentation schemes," in *Proc. Int. Database Eng. Appl. Symp. (IDEAS)*, Aug. 1999, 408-414, doi: 10.1109/IDEAS.1999.787292.
[50] M. Kvet and K. Matiaško, "Data loading and migration methods in the cloud environment," in *Proc. Commun. Inf. Technol. (KIT)*, Oct. 2021, pp. 1–6.
[51] R. Cornejo, *Dynamic Oracle Performance Analytics*. 2018, doi: 10.1007/978-1-4842-4137-0.
[52] M. Kvet, "Relational data index consolidation," in *Proc. 28th IEEE Conf. Open Innov. Assoc. (FRUCT)*, Jan. 2021, pp. 215–221, doi: 10.23919/FRUCT50888.2021.9347614.
[53] A. Dudáš, P. Voštinár, J. Skrinarova, and J. Siláči, "Improved process of running tasks in the high performance computing system," in *Proc. 16th Int. Conf. Emerg. eLearning Technol. Appl. (ICETA)*, Nov. 2018, pp. 133–140, doi: 10.1109/ICETA.2018.8572230.
[54] M. Kvet, "Autonomous temporal time zone management," in *Proc. 47th Annu. Conf. IEEE Ind. Electron. Soc. (IECON)*, Oct. 2021, pp. 1–6, doi: 10.1109/IECON48115.2021.9589547.
[55] M. Kvet, "Autonomous temporal transaction database," in *Proc. 30th Conf. Open Innov. Assoc. (FRUCT)*, Oct. 2021, pp. 121–128, doi: 10.23919/FRUCT53335.2021.9599977.
[56] Z. Qian, J. Wei, Y. Xiang, and C. Xiao, "A performance evaluation of DRAM access for in-memory databases," *IEEE Access*, vol. 9, pp. 146454–146470, 2021, doi: 10.1109/ACCESS.2021.3123379.