UNIVERSITY OF ŽILINA

FACULTY OF MANAGEMENT SCIENCE AND INFORMATICS

INDEXING, DEPLOYMENT AND SEARCHING ALGORITHMS IN LARGE DATABASES

DISSERTATION THESIS

28360020223001

Study programme:	Applied Informatics
Field of study:	Informatics
Department:	Department of Informatics
	Faculty of Management Science and Informatics, University of Žilina
Supervisor:	doc. Ing. Michal Kvet, PhD.

Žilina, 2022 **Ing. Veronika Šalgová**

ABSTRAKT

ŠALGOVÁ, Veronika: *Algoritmy indexovania, rozmiestňovania a vyhľadávania v rozsiahlych databázach* [dizertačná práca] – Žilinská univerzita v Žilina. Fakulta riadenia a informatiky; Katedra informatiky. – Školiteľ: doc. Ing. Michal Kvet, PhD. – Stupeň odbornej kvalifikácie: Doktor filozofie v študijnom odbore Informatika. Žilina: FRI ŽU v Žiline, 2022.

Dizertačná práca sa zaoberá problematikou indexovania, rozmiestňovania a vyhľadávania v rozsiahlych databázach. Množstvo dát v dnešnej dobe rapídne rastie, čo prináša značné výzvy. Úložný priestor sa stáva dôležitým ukazovateľom nákladov. Veľký dôraz sa kladie na zrýchľovanie prístupu k dátam. Výkon sa často zlepšuje pomocou indexovania, vytvárania partícií, kompresie údajov v tabuľkách alebo indexoch alebo vhodnej správy úložného priestoru. V našej dizertačnej práci sme sa zaoberali metódami prístupu k indexom, procesom prístupu k dátam, ich optimalizáciou a naším prístupom k odstráneniu potreby prehľadávania celej tabuľky metódou Table Access Full. Štrukturálny index označený ako master sa používa na prístup k záznamu a jeho lokalizáciu s dôrazom na možnosti fragmentácie. Zamerali sme sa aj na znižovanie času prístupu k dátam prostredníctvom rôznych techník vytvárania partícií a indexovania a ich kombinácií, čo prinieslo výrazné zmeny vo výkonnosti. Vplyv použitých metód a techník bol skúmaný na základe času prístupu k údajom, nákladov na CPU a času vykonávania jednotlivých operácií, akými sú Insert, Update a Delete. Zamerali sme sa aj na vplyv kompresie údajov v tabuľkách, indexoch a ich kombináciách, ktorý bol určovaný na základe času prístupu k údajom a nákladov na CPU. Zaoberali sme sa taktiež automatickým vyvažovaním indexov, kontrolou nedefinovaných hodnôt, identifikáciou migrovaných riadkov a vyhodnotením vhodnosti použitých indexov.

Kľúčové slová: Databázový systém. Indexové štruktúry. Vytváranie partícií. Kompresia dát. Vyvažovanie indexov.

ABSTRACT

ŠALGOVÁ, Veronika: *Indexing, Deployment and Searching Algorithms in Large Databases* [dissertation thesis] – University of Žilina. Faculty of Management Science and Informatics; Department of Informatics. – Supervisor: doc. Ing. Michal Kvet, PhD. – Qualification level: Philosophiae doctor in the study field Informatics. Žilina: FRI ŽU in Žilina, 2022.

The dissertation thesis deals with the issue of indexing, deployment and searching in large databases. The amount of stored data is growing rapidly, and it brings considerable challenges. The enormous growth in the volume of data makes storage one of the biggest cost elements. Great emphasis is placed on the improvement of fast access to data. Performance improvements are often provided through the use of indexing, partitioning, compression of data in tables or indexes, or appropriate storage management. In our dissertation thesis, we dealt with index access methods, the process of the data access, its optimization, and our approach to removing the need to search the entire table physically by using the Table Access Full method. The structural index denoted as the master is used to access and locate a record with an emphasis on fragmentation options. We also focused on reducing data access time through various partitioning and indexing techniques and their combinations, which brought significant changes in performance. The impact of the methods and techniques used was researched on the basis of data access time, CPU costs, and execution time of individual operations such as Insert, Update, and Delete. We also focused on the effect of data compression in tables, indexes, and their combinations, which was determined based on data access time and CPU costs. We dealt with automatic index balancing, control of undefined values, migrated rows identification, and index structure efficiency evaluation.

Keywords: Database System. Index Structures. Partitioning. Data Compression. Index balancing.

CONTENTS

ABSTRAKT 2

ABSTRACT 3

CONTENTS 4

1 INTRODUCTION 8

2 DATABASE SYSTEMS 10

2.1 DISTRIBUTED DATABASE SYSTEMS 10

2.2 RELATIONAL DATABASE SYSTEMS 11

2.2.1 ORACLE DATABASE INFRASTRUCTURE 13

2.3 DATA WAREHOUSES 16

2.3.1 DATA WAREHOUSE VS. DATABASE 17

2.4 OBJECT-RELATIONAL DATABASE SYSTEMS 17

2.5 DATA MARTS 17

2.5.1 DATA MART VS. DATA WAREHOUSE 18

2.5.2 TYPES OF DATA MARTS 19

2.5.3 BENEFITS OF DATA MARTS 19

3 SEARCHING IN DATA STRUCTURES 21

3.1 SEQUENTIAL SEARCH 21

3.2 BINARY SEARCH 21

3.3 BINARY SEARCH TREE 22

3.4 2-3 TREE 22

3.5 2-4 TREE 23

4 INDEX 25

4.1 B-TREE INDEX 26

4.1.1 INDEX-ORGANIZED TABLE 33

4.1.2 UNIQUE INDEX 34

4.1.3 REVERSE KEY INDEX 34

4.1.4 COMPOSITE INDEX 35

4.1.5 KEY COMPRESSED INDEX 35

4.1.6 DESCENDING INDEX 36

4.2 BITMAP INDEX 36

4.3 FUNCTION-BASED INDEX 39

4.4 PREFIX INDEX ... 39
4.5 VIRTUAL INDEX ... 40
4.6 INVISIBLE INDEX ... 40
4.7 HASH INDEX ... 41
4.8 PARALLEL AND NOLOGGING CLAUSE ... 42
4.9 AUTOMATIC INDEXING ... 44
4.10 INDEX SCAN METHODS ... 47
4.10.1 INDEX RANGE SCAN ... 47
4.10.2 INDEX UNIQUE SCAN ... 47
4.10.3 FULL INDEX SCAN ... 48
4.10.4 FAST FULL INDEX SCAN ... 48
4.10.5 INDEX SKIP SCAN ... 48
4.11 JOIN STRATEGIES ... 48
4.12 SHRINKING THE SPACE ... 50
4.13 FULL-TEXT SEARCH ... 51
5 PARTITIONING ... 53
5.1 PARTITIONING TECHNIQUES ... 54
5.1.1 HASH PARTITIONING ... 55
5.1.2 RANGE PARTITIONING ... 55
5.1.3 LIST PARTITIONING ... 56
5.2 PARTITIONING METHODS ... 57
5.2.1 SINGLE-LEVEL PARTITIONING ... 57
5.2.2 COMPOSITE PARTITIONING ... 58
5.3 PARTITIONING SCHEME ... 58
5.4 PARTITIONED INDEX ... 59
5.5 PARTITIONING TO IMPROVE PERFORMANCE ... 62
6 AUTOMATIC STORAGE MANAGEMENT ... 63
7 CURRENT RESEARCH PERSPECTIVES ... 64
7.1 MASSACHUSETTS INSTITUTE OF TECHNOLOGY ... 64
7.2 TECHNICAL UNIVERSITY OF MUNICH ... 67
7.3 UNIVERSITY OF ALBERTA ... 69
7.4 KARUNYA INSTITUTE OF TECHNOLOGY AND SCIENCES ... 70
8 OWN CONTRIBUTION ... 72
8.1 MASTER INDEX ... 72

8.1.1 MASTER INDEX DEFINITION 74
8.1.2 INDEX MASTER METHOD 75
8.1.3 ROWID VS. BLOCKID 78
8.1.4 RESULTS 79
8.1.5 SUMMARY 82
8.2 REDUCING DATA ACCESS TIME USING PARTITIONING 83
8.2.1 RESULTS 83
8.2.2 RANGE PARTITIONING 83
8.2.3 LIST PARTITIONING 85
8.2.4 SUMMARY 87
8.3 EFFECT OF PARTITIONING AND INDEXING 88
8.3.1 METHODOLOGY 88
8.3.2 RESULTS 90
8.3.3 SUMMARY 95
8.4 EFFECT OF INDEXES ON DML OPERATIONS 96
8.4.1 DATA 96
8.4.2 RESULTS 96
8.4.3 SUMMARY 99
8.5 THE IMPACT OF TABLE AND INDEX COMPRESSION 99
8.5.1 DATA COMPRESSION 100
8.5.2 COMPRESSED VS. UNCOMPRESSED BLOCK 101
8.5.3 COMPRESSED TABLE 102
8.5.4 METHODOLOGY 104
8.5.5 RESULTS 105
8.5.6 SUMMARY 106
8.6 THE COMPLEXITY OF THE DATA RETRIEVAL PROCESS USING THE INDEX EXTENSION 107
8.6.1 MIGRATED ROWS IDENTIFICATION AND REFLECTION 108
8.6.1.1 MIGRATED ROWS – SOLUTION 1 109
8.6.1.2 MIGRATED ROWS – SOLUTION 2 110
8.6.2 MIGRATION-POST-APPLICATION 111
8.6.3 STORING PATH 112
8.6.4 DATA POINTER REFLECTION 113
8.6.5 PRIORITY MANAGEMENT 114

8.6.6 ARCHITECTURE ENHANCEMENTS 120
8.6.7 METHODOLOGY 121
8.6.8 RESULTS 123
8.6.8.1 MIGRATED ROW LIMITATION 123
8.6.8.2 PRIORITY MANAGEMENT 128
8.6.8.3 INDEXING OUTSIDE THE MAIN TRANSACTION 132
8.6.8.4 DATA COMPLEXITY AND RELIABILITY 133
8.6.9 SUMMARY 136
9 CONCLUSION 138
LIST OF OWN PUBLICATIONS 142
REFERENCES 144

1 INTRODUCTION

Storing and managing collections of data is a significant part of information technologies. The modern era of computerization brings an explosion of information which need to be stored and processed. They are a very important part of many information systems, from commercial systems, through technical and technological systems, the web, and mobile applications to the management of scientific data in various fields. Relational database systems cover the main part of the current data management of information technology. A large amount of data requires a lot of storage space, so very large databases with huge amounts of data are coming to the foreground. Data is organized in the process of database normalization reducing redundancy and dependency of information. This data is subsequently used for query execution and gaining the desired outputs.

Fast access to data is becoming increasingly important today and great emphasis is placed on improving it. However, with a large amount of data, their processing usually takes a long time, so in our work, we focus on the analysis and implementation of various extensions that ensure improved performance and faster access to data.

Our work is divided into nine main chapters. The chapter following the introduction deals with database systems, their different types, similarities, advantages, and disadvantages. The third chapter focuses on searching in data structures. It describes the different types of searching and tree data structures. The fourth chapter is devoted to the indexes. It contains an analysis of different types of indexes, indexing methods, methods used for scanning indexes, and joining tables. Partitioning is discussed in the fifth chapter. It describes the analysis of various partitioning methods, techniques, and ways of partitioning the indexes. The sixth chapter deals with automatic storage management and its components and technologies. In the seventh chapter, we analyze the institutes and universities that deal with similar issues and emphasize some of their research. In the eighth chapter, we deal with our own contribution, which focused mainly on the impact of indexes, partitioning, and compression on data access time, CPU costs, and DML operation times, but also on control of undefined values, migrated rows identification, automatic index balancing, and index structure efficiency evaluation.

We have used the Oracle database system to evaluate the results. We chose this database system due to deeper cooperation with this company in our department, even within projects related to the Oracle database system, such as CodeIn (https://code-in.org/) and BeeApex (https://beeapex.eu/my/). However, our proposed solutions are universal and can be applied to other database systems. We tested some of the solutions in the MySQL, Postgres, and MS SQL database systems as well.

In the conclusion, the contribution of our work is summarized both for practice and for the field of applied informatics. In addition, we outline scientific problems that need to be addressed in further research in this area.

2 DATABASE SYSTEMS

The complexity and scope of information systems are constantly increasing. This creates a greater need to make the manipulation of information as simple, and fast as possible. There is also a need for the functions that provide support and access to data to be sufficiently efficient, fast, and robust. The basic technology that should ensure the quality operation of the entire system is database technology. It is a unified set of concepts, means, and techniques used to create information systems. Today's database systems are part of almost every information system, from commercial systems, through technical and technological systems, web, and mobile applications to the management of scientific data in many fields.

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 a set of 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. Data cannot be updated directly in the database but must be memory-loaded in a block granularity for the evaluation and processing [8] [76].

The term *Database System* (DBS) refers to a set of interrelated data along with software that allows access to the data. The database system can also be understood as a computer system for managing stored records.

2.1 DISTRIBUTED DATABASE SYSTEMS

A set of computer network nodes is called *a distributed database system* if these nodes are interconnected in a communication network, where each node is a separate database system, but these nodes cooperate with each other in such a way that each node can access data stored on another node as if they were placed on their own node [56].

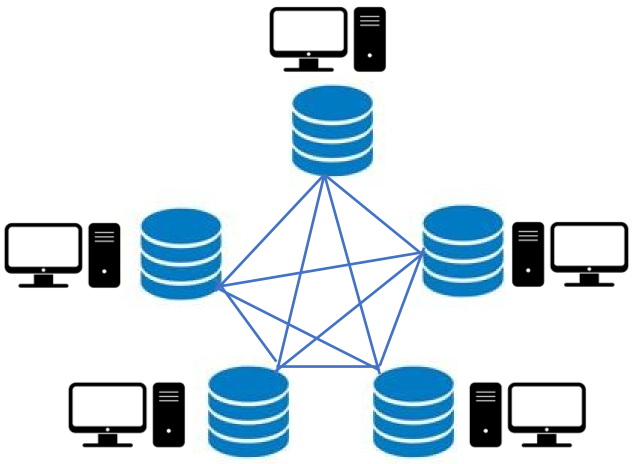


Fig. 2.1: Architecture of a distributed database system

2.2 RELATIONAL DATABASE SYSTEMS

The main part of the current data management of information technology is covered by *relational database systems*. Data are formed into the relations connected using relationships. They are represented in tables, in which each row is a record with a unique ID. Each tuple is physically stored in the database operated by the background processes of the instance [57]. The user query is transferred to the server, analyzed, and processed. Data are sent back to the user as the result set. The main part of the processing and query evaluation is just access to the data themselves [26] [38].

The transaction is the main property of the relational paradigm. Any change on the data (insert, update, or delete statement) is a component of the transaction, which can be consequently accepted or refused. Thus, before data is visible publicly, it must be approved – committed [18]. Transactions ensure the complex correctness and reliability of the data. They are important not only for the data changes but the *Select* queries, as well. From the physical point of view, data are always accessed from the *Buffer cache* memory structure [52]. In an optimistic case, the required data may be directly available, whereas they are present in the memory. Thus, the result set is constructed and sent to the user. If the data is not present there, or some portion of them is missing, they must be loaded from the physical database storage to the memory, where the next processing steps are executed. Physical database storage is delimited by the data files belonging to the tablespaces. Internally, each data file is made up of individual blocks of the same size. During the processing, the whole block is transferred into the memory, where the relevant data are searched and located. Selection of the block with relevant data can be done using two techniques – sequential scanning or using an index [17] [37].

A relational database offers complexity, robustness, and security of the data stored inside. The core element differentiating the database and file system is just the transaction support. Its management ensures the data is reliable passing all requirements and constraints. Transaction theory defines four aspects of dealing with the databases – atomicity, consistency, isolation, and durability. *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 taking emphasis on 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, as well. The third aspect is *isolation*. Changes done inside the transaction are made visible just after its approval by spreading the operations to the whole environment. Finally, the *durability* covered by the logging ensures, that the approved transaction changes are visible and will be present in the system even after the system crash [9].

Query processing consists of several steps, which are consecutively executed. The output of the individual step is transferred as the input of the subsequent one. Figure 2.2 shows the query processing steps.

Parser performs *syntactic* (command grammar) and *semantic* (object existence and access rights) analyzing of the query and rewriting the original plan to a set of relational algebra operations. *Optimizer* suggests the most effective way to get query results, based on the optimization methods, developed indexes, and collected statistics. Thus, it selects the best (suboptimal) query execution plan, which is used in the next *Row source generator* step. It creates the *execution plan* for the given SQL query in the form of a tree, whose nodes are made up of individual row sources. Afterward, the SQL query is executed with emphasis on the provided execution plan. The result set is constructed and sent to the client [20] [35]. The most important step in terms of the processing efficiency, optimization, and access rules is just the *execution plan*, which determines the usage of indexes [48].

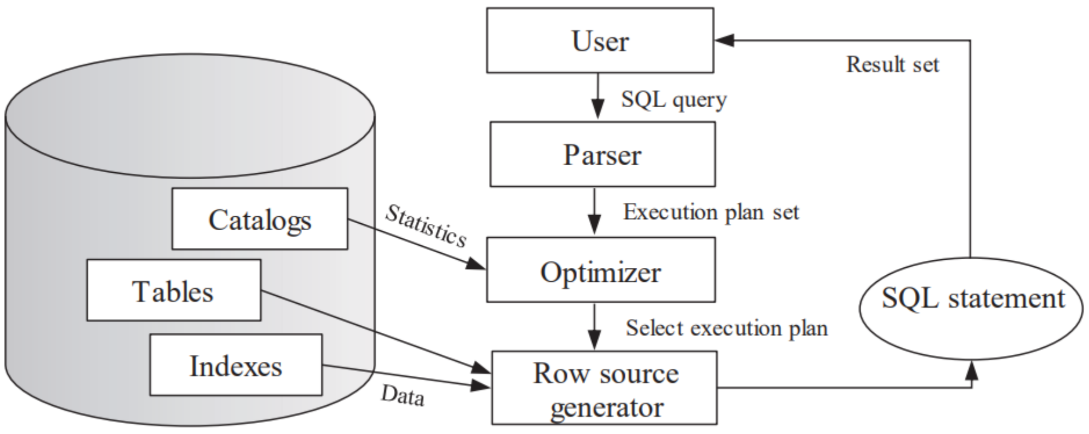
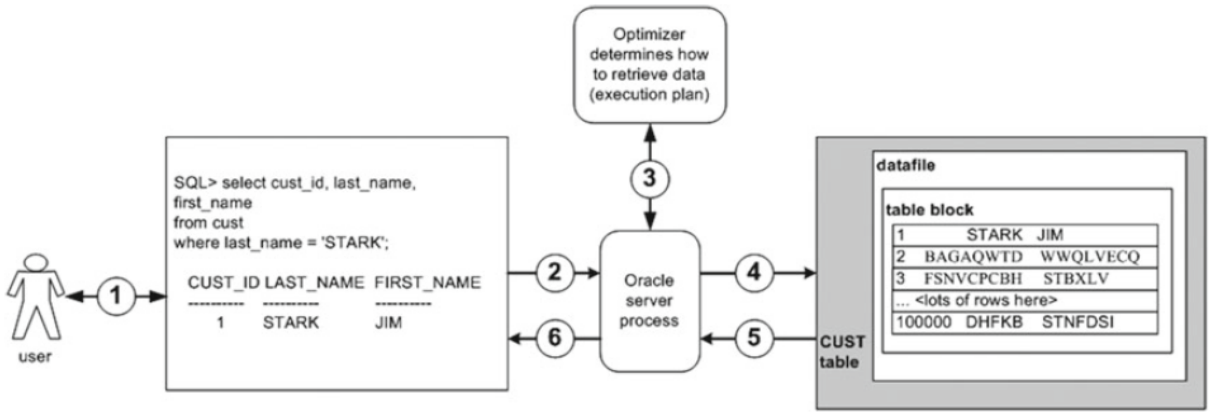


Fig. 2.2: SQL statement evaluation

Figure 2.3 shows the data query processing principles from the user and server perspectives. 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. By using the shared system, multiple client sessions can be routed to the common server process, which shares the resources.



```
SQL> select cust_id, last_name,
first_name
from cust
where last_name = 'STARK';

 CUST_ID LAST_NAME  FIRST_NAME
 -------- ---------  ----------
     1    STARK      JIM
```

```
1         STARK   JIM
2   BAGAQWTD   WWQLVECQ
3   FSNVCPCBH    STBXLV
... <lots of rows here>
100000  DHFKB   STNFDSI
```

Fig. 2.3: Data query processing principles

2.2.1 ORACLE DATABASE INFRASTRUCTURE

The physical Oracle database infrastructure looks different on various operating systems but differs from other database systems, as well. Oracle database system consists of the instance, database, and optimally, container and pluggable databases are covered, supervised by the real application cluster (RAC) environment [40]. There are two basic Oracle architectures – single- and multi-tenant architecture.

A single-tenant architecture 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. 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 balancing of the workload across individual instances registered. Each instance node has a separate listener, processes, and memory [15].

In March 2017, a new architecture was created and offered by the Oracle 12c version. A *multi-tenant container database* (root container database) consists of a set of 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, which 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. The client connects to the server by the listener, which creates the server process in the container instance. 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, in comparison with single tenancy, environment characteristics are stored just once for the whole container [62]. Figure 2.4 shows the container database with two pluggable databases attached.

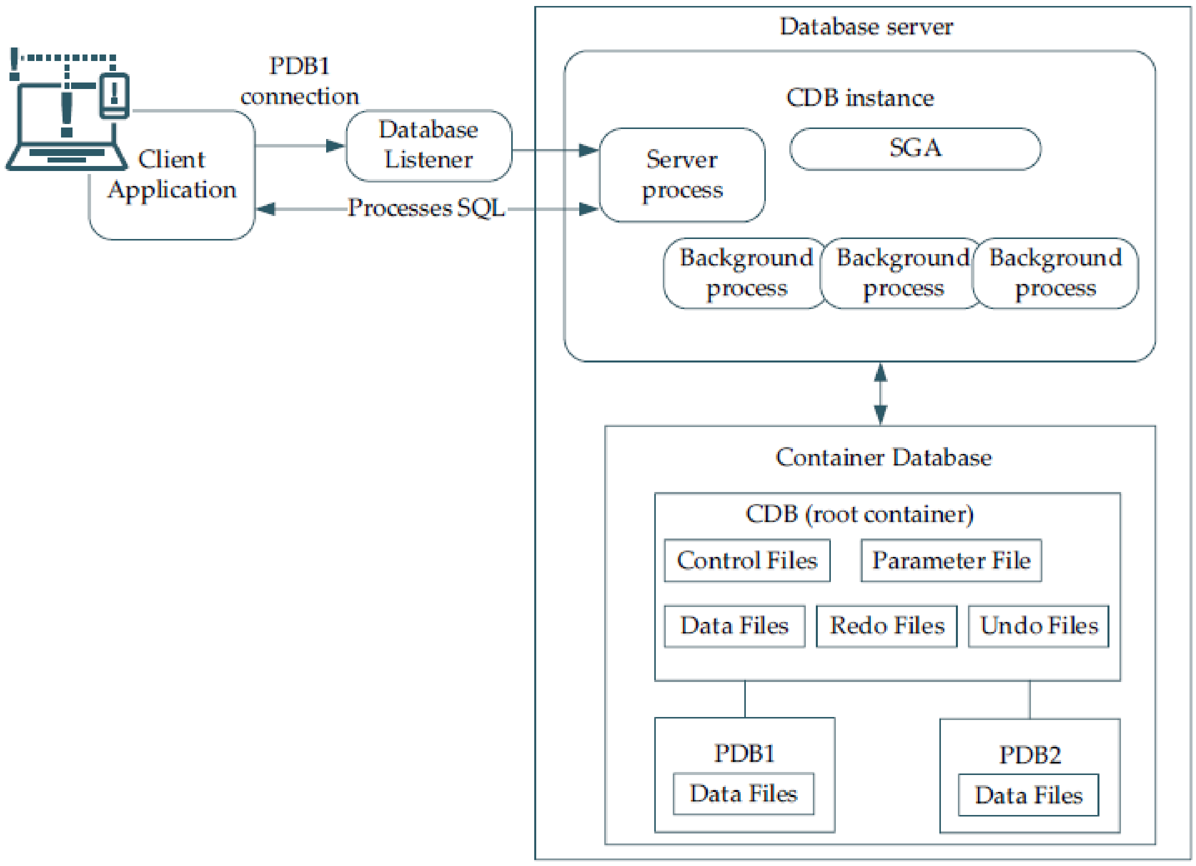


Fig. 2.4: Container database architecture

The general solution is provided by Multitenant RAC Database, where multiple node instances can be connected to one container database operating multiple pluggable databases. Compared with the architecture presented in Figure 2.4, there are numerous instances with separate memory and process structure. SCAN listeners are connected to the individual listeners for each instance evaluating the workload to ensure proper performance. This solution generally provides performance complexity, robust scalability, and error-prone management. Interconnection between the physical database and instance is done by the container, if any instance fails, the workload is automatically routed to the survivor ensuring the availability and reliability of the whole solution model [40].

From the performance perspective, the relevant model is also a S*harded database* introduced in 2017 [1], providing linear scalability, fault tolerance, and geographic data distribution by using horizontal fragmentation across multiple regions [62]. It is shown in Figure 2.5. Each partitioned database has its own instance forming the sharded database, which is connected to the connection pool for the client connections supervised by the shard directors.

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 [1].

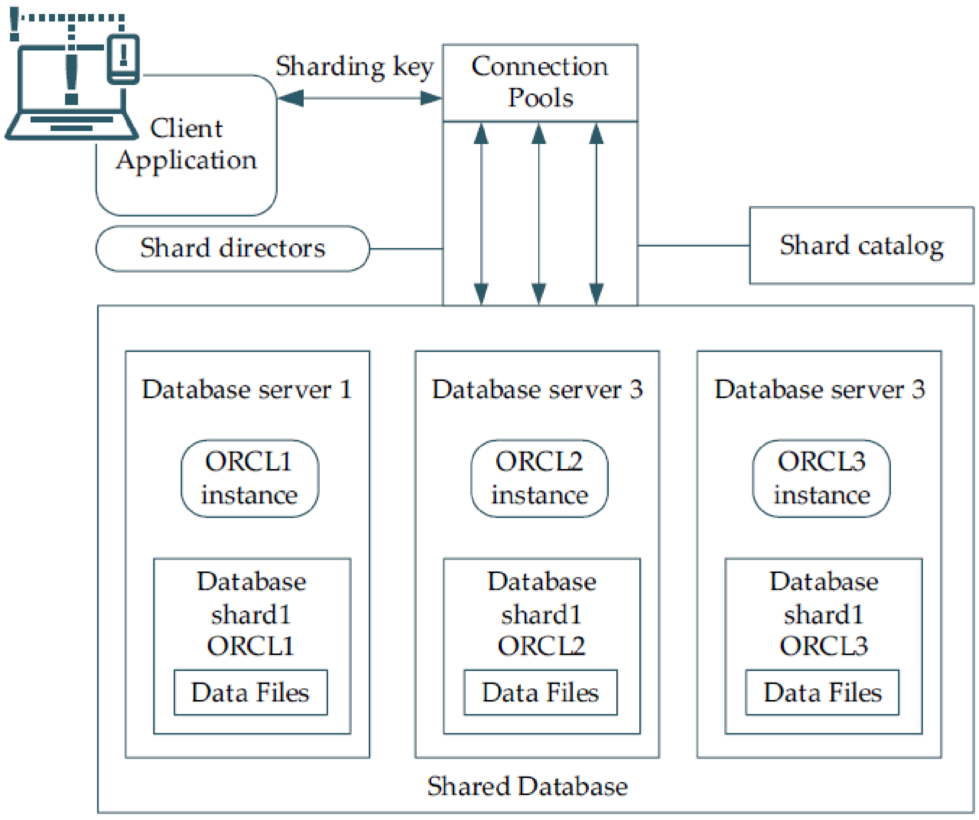


Fig. 2.5: Sharded database

2.3 DATA WAREHOUSES

A data warehouse is a subject-oriented, integrated, nonvolatile, and time-variant collection of data in support of management's decisions. The data warehouse contains granular corporate data. It is a system that aggregates data from different sources into a single, central, consistent data store to support business analytics, data mining, artificial intelligence (AI), and machine learning. A data warehouse enables an organization to run powerful analytics on huge volumes (petabytes and petabytes) of historical data in ways that a standard database cannot. Of all the aspects of a data warehouse, integration is the most important. Data is fed from multiple disparate sources into the data warehouse. As the data is fed it is converted, reformatted, resequenced, summarized, and so forth [30] [35].

2.3.1 DATA WAREHOUSE VS. DATABASE

While a database captures and stores data from a single (usually current) point in time, a data warehouse encompasses current and historical data required for predictive analytics, machine learning, and other advanced analysis. A database is built primarily for fast queries and transactional processing, not analytics. A database typically serves as the focused data store for a specific application, whereas a data warehouse stores data from any number (or even all) of the applications in an organization [35].

The main differences between databases and data warehouses are described below:

DATABASE	DATA WAREHOUSE
Designed to **record** data.	Designed to **analyze** data.
Stores **detailed** data.	Stores **summarized** data.
Uses Online Transactional Processing (**OLTP**).	Uses Online Analytical Processing (**OLAP**).
Performs fundamental business operations and transactions.	Allows users to analyze business data.
Data is available in **real-time**.	Data **must be refreshed** when needed.
Application-oriented data collection.	**Subject-oriented** data collection.
Limited to a single application.	Draws data from a range of other applications.

Tab. 2.1: Differences between Databases and Data Warehouses [2]

2.4 OBJECT-RELATIONAL DATABASE SYSTEMS

These systems are composed of relational database systems and object-oriented database systems. Unlike relational database systems, they are also enriched by supporting the basic components of object-oriented database systems in their schemas and a query language, such as objects, classes, and inheritance. They connect relational databases and object-oriented modeling techniques [79].

In object-relational database systems, data formats such as XML and JSON can be used. There is a big advantage that both of them can be indexed [29].

2.5 DATA MARTS

A *data mart* is a subset of a data warehouse focused on a particular line of the business, department, or subject area. It makes specific data available to a defined group of users, which allows them to quickly access critical insights without searching within a more complex data warehouse or manually aggregating data from different sources. Because a data mart only contains the data applicable to a certain business area, it is a cost-effect way to gain actionable insights quickly. For example, many companies may have a data mart that aligns with a specific department in the business, such as finance, sales, or marketing [64].

2.5.1 DATA MART VS. DATA WAREHOUSE

Data marts and data warehouses are both highly structured repositories where data is stored and managed until it is needed. However, they differ in the scope of data stored. Data warehouses are built to serve as the central store of data for the entire business, whereas a data mart fulfills the request of a specific division or a business function. Because a data warehouse contains data for the entire company, it is best practice to have strict control over who can access it. What is more, querying the needed data in a data warehouse is a very difficult task for the business. Thus, the primary purpose of a data mart is to isolate or partition a smaller set of data from a whole to provide easier data access for the end consumers [35].

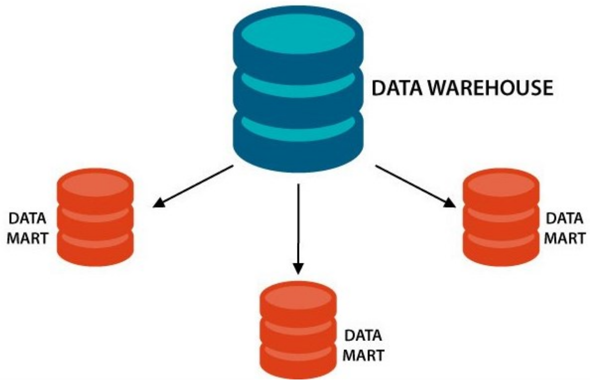


Fig. 2.6: Data Warehouse vs. Data Mart

A data mart can be created from an existing data warehouse, which is the top-down approach, or from other sources, such as internal operational systems or external data. Similar to a data warehouse, it is a relational database that stores transactional data in columns and rows making it easy to organize and access. On the other hand, separate business units may create their own data marts based on their own data requirements. If business needs dictate, multiple data marts can be merged together to create a single data warehouse [30].

The main differences between data marts and data warehouses are described below:

	DATA MART	DATA WAREHOUSE
SIZE	Less than 100 GB	More than 100 GB
SUBJECT	Single subject	Multiple Subjects
SCOPE	Line-of-Business	Enterprise-wide
DATA SOURCES	Few sources	Many Source Systems
DATA INTEGRATION	One Subject Area	All Business Data
TIME TO BUILD	Minutes, weeks, months	Many months to years

Tab. 2.2: Differences between Data Marts and Data Warehouses

2.5.2 TYPES OF DATA MARTS

The types of data marts are categorized based on their relationship to the data warehouse and the data sources that are used to create the system. Data marts can be dependent, independent, and hybrid.

Dependent data marts are created from an existing enterprise data warehouse. It begins with storing all business data in one central location. The newly created data marts extract a clearly defined subset of the primary data whenever required for analysis.

Independent data marts act as a stand-alone system that does not rely on a data warehouse. Analysts can extract data on a particular subject or business process from internal or external data sources, process it, and then store it in a data mart repository until needed for business analytics. Independent data marts are not difficult to design and develop. They are beneficial to achieve short-term goals but may become cumbersome to manage as business needs expand and become more complex.

Hybrid data marts combine data from existing data warehouses and other operational source systems. It unites the speed and end-user focus of a top-down approach with the benefits of the enterprise-level integration of the bottom-up method [73].

2.5.3 BENEFITS OF DATA MARTS

Managing big data and gaining valuable business insights is a challenge all companies face, and one that most are answering with strategic data marts. A data mart is a **time-saving solution** for accessing a specific set of data for business intelligence. It can be an **inexpensive alternative** to developing an enterprise data warehouse, where required data sets are smaller. An independent data mart can be up and running in a week or less. Dependent and hybrid data marts can **improve the performance** of a data warehouse by taking on the burden of processing, to meet the needs of the analyst. When dependent data marts are placed in a separate processing facility, they significantly reduce analytics processing costs as well. Other advantages of a data mart include **data maintenance**, when different departments can own and control their data, **simple setup** because the simple design requires less technical skill to set up, **analytics** as the key performance indicators can be easily tracked, and **easy entry**, in that data marts can be the building blocks of a future enterprise data warehouse project [30].

3 SEARCHING IN DATA STRUCTURES

The process of finding the location of an element in a list is one of the important parts of many algorithms. The efficiency of searching an element increases the efficiency of any algorithm. The most famous techniques of searching in data structures are *sequential search* and *binary search*.

In database systems, the structure is managed at the block level. When searching for a record, it is always necessary to work with the entire block. It is not possible to load only one record. Also, when inserting records, even if the block is empty, it is necessary to copy the entire block.

3.1 SEQUENTIAL SEARCH

The traditional technique for searching for an element in a collection of elements is the sequential search. All the elements of the list are traversed one by one to find if the element is present in the list or not. An example of this type of algorithm is a linear search. As a linear search algorithm does not use any extra space thus its space complexity is O(n) for an array of n number of elements. When the element to search is not present in the array there occurs the worst-case complexity with the value O(n). When the first element is the element to be searched there occurs the best-case complexity with value O(1). The average time complexity is O(n).

Sequential data scanning is demanding, and too time consuming, pointing to the efficiency of the whole process and the performance. It is not related just to the data retrieval itself, other operations like Update needs to locate existing data tuples to be updated. Similarly, when dealing with 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 [79].

3.2 BINARY SEARCH

The technique to search an element in the list using the divide and conquer technique is a binary search. It is a very fast and efficient searching technique. It directly goes to the middle element of the list and divides the array into two parts and decides in which sub-array the element exist. The worst-case complexity of the binary search is O(n $\log_n$), the best-case complexity is O(1) and the average-case complexity is O(n $\log_n$).

3.3 BINARY SEARCH TREE

A binary search tree (BST) is a node-based data structure in which each node has at most two children referred to as the left child and the right child. Each node contains a value from a well-ordered set and has the following properties:

- The left subtree of a node contains only nodes with a key-value that is smaller than the key-value of the node.
- The right subtree of a node contains only nodes with a key-value that is larger than the key-value of the node.
- The left and the right subtree each must also be a binary search tree.

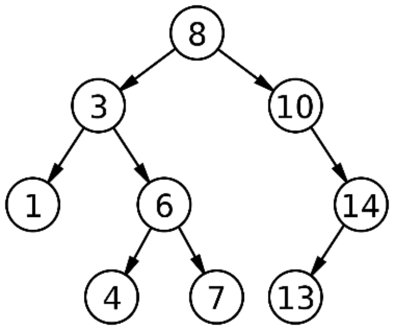


Fig. 3.1: An example of a binary search tree

Searching in a BST is started from the root node. A binary search method is used, so if the searched data is smaller than the key value, searching continues in the left subtree. Otherwise, if the data is larger than the key value, searching continues in the right subtree.

A BST is fast in inserting and deleting data when it is balanced. However, the shape of a BST depends upon the order of insertion, and it can degenerate. Searching in a BST takes a long time [60].

3.4 2-3 TREE

A tree data structure, in which every internal node is a 2-node, or a 3-node is called a 2-3 tree. A 2-node has one element and two children, and a 3-node has two elements and three children. Leaf nodes can heave one or two data elements. This type of tree is balanced, so each leaf is at the same level. Data in a 2-3 tree are kept in a sorted manner.

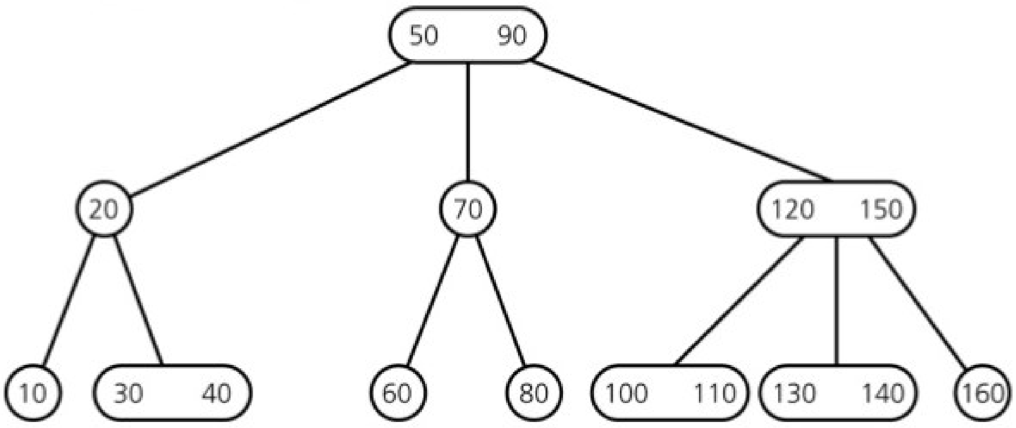


Fig. 3.2: An example of a 2-3 tree

Searching for an item in a 2-3 tree is similar to searching for an item in a binary search tree. Searching starts in a root node and continues to the correct subtree according to the key value.

Even though searching a 2-3 tree is not more efficient than searching a binary search tree (BST), by allowing the node to have three children, a 2-3 tree might be shorter than the shortest possible BST. It is easier to maintain the balance of a 2-3 tree than of a BST after a sequence of insertions [34].

3.5 2-4 TREE

A tree structure called a 2-4 tree, or a 2-3-4 tree is a generalization of a 2-3 tree. It has three types of nodes. A 4-node has three elements and four children, a 3-node has two elements and three children, and a 2-node has one element and two children. This type of tree is balanced and has data kept in a sorted manner.

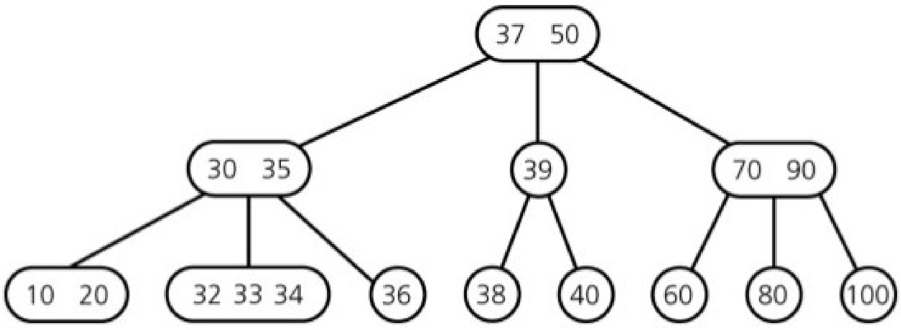


Fig. 3.3: An example of a 2-4 tree

There is an advantage of 2-4 trees over 2-3 trees. Insertion and deletion can be performed by a single root-to-leaf pass rather than by a root-to-leaf pass followed by a leaf-to-root pass.

Despite the mentioned advantages, none of the above-mentioned trees is suitable for use in databases. Among several structures, it is most advantageous to use B-trees, resp. B+ trees that can provide fast data inserting and deleting. This type of tree is described in the following chapter.

4 INDEX

An important and powerful part of the optimization of query processing is *an index structure* that can significantly improve the performance of the database. Scanning through very large tables consisting of millions or billions of rows is a big waste of time when only a few of them need to be returned. An index is a database object which can be created optionally and is used primarily to increase query performance and reduce processing time, but also CPU, I/O, and other system resources. Its purpose is similar to a book index that associates each topic with the page numbers on which the topic is located. Thanks to this information it is easy to navigate directly to a page containing a specific topic. The number of pages to read is minimal if the topic appears only on a few pages. The more the topic appears in the book, the less useful the index is.

The database index is used for direct access to the column value of a row inside the database with a row identifier called ROWID. The ROWID consists of these layers: identification of the data file, in which the row resists, the pointer to the block, and position inside it. It also uses the specific object identifier. Thus, based on the definition, the ROWID value is unique for the standalone database. With ROWID, table data can be easily retrieved with a minimum number of reads executed. However, if there exists no suitable index, sequential data scanning, represented by the block-by-block memory loading is necessary. Such activity has, however, a significant impact on the 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 all rows evaluation. The problem can cause block fragmentation, by which the loading efficiency is lowered [63]. Block data amount can be significantly higher in comparison with optimal storage inside the blocks [28]. Moreover, data blocks are not created and associated with the table separately, instead, the set of the blocks is allocated at once, forming the data extent. As a result, even completely free blocks can be identified. Despite the fact, that the block does not hold any data, it is memory-loaded, whereas the system does not hold references to the empty blocks, at all. The *Table Access Full* method is executed, which is one of the most expensive operations [19] [31].

The suitability of the index is crucial for the processing, whereas the order of the attributes delimits the index structure. The order of the elements inside the index should reflect the query conditions and optionally values listed in the Select clause. If all necessary attributes are inside the index, but in a non-suitable format, sequential scanning must be done.

Either scanning the whole table or the index. As a result, multiple index sets are defined in the system to ensure performance [7].

Although indexes bring a significant improvement in database performance, it is not appropriate to create them on every column or combination of columns of all the tables. Indexes consume disk space and system resources. When data in tables are updated, corresponding indexes need to be changed as well. It means they use storage, I/O, CPU, and memory resources [41]. For the Select statement, performance is better due to the use of an index. However, when using the Insert, Update, and Delete statements, performance slows down. The unnecessarily large use of indexes can markedly decrease the performance of the database. The creation and deletion of an index absolutely do not affect the data in a table, it affects only the query performance.

In the database systems, there exist various index structures and approaches.

4.1 B-TREE INDEX

The default and most often used type of index is *B-tree*, respectively *B+ tree*. This structure consists of a tree in which each path from the root to the leaf has the same length [35]. It means the tree is balanced. It is optimized for systems that write and read a lot of data. The model of the B-tree structure is shown in Figure 4.1. Locators of the rows in the physical database, ROWID values, are stored in the leaf layer.

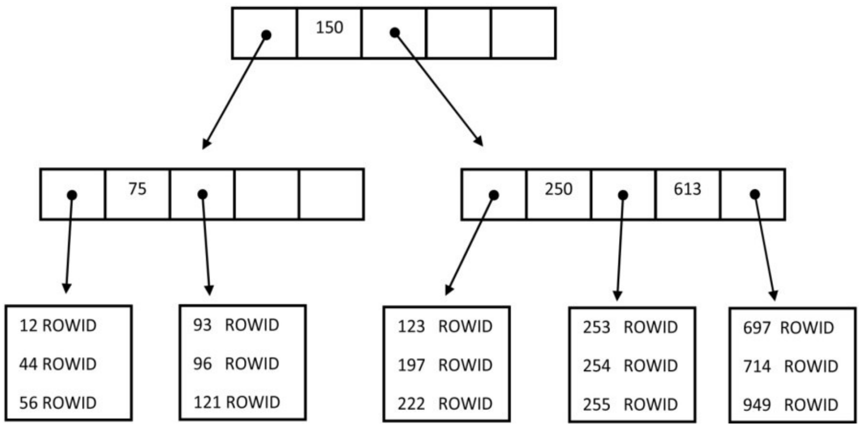


Fig. 4.1: B-tree structure

The B-tree provides ordered sequential access to the index. Iterations can be executed over the items or keys in ascending or descending order. B+ tree index structure varies in the leaf layer, in which individual nodes are chained together and form a linked list [53]. This layer contains sorted data based on the attributes which are indexed there. The B-tree index is suitable for the high cardinality columns or expressions.

An index of the B-tree structure is created using *CREATE INDEX* statement as follows:

```
CREATE INDEX ind_student
  ON student(surname);
```

B-tree indexes have several subtypes: index-organized table, unique index, reverse key index, key compressed index, and descending index [44].

An empty index corresponding to an empty table basically consists of one empty block. The index has a *BLEVEL*, which is the number of branch levels, of 0 and a *HEIGHT* of 1. The value can be obtained from *DBA_INDEXES* as follows:

```
SELECT INDEX_NAME, BLEVEL
  FROM DBA_INDEXES
    WHERE TABLE_NAME = <table_name>;
```

The height value can be found in the *INDEX_STATS* view after the index has been analyzed using the command:

```
ANALYZE INDEX <index_name> VALIDATE STRUCTURE;
```

The only block is the root block of the index and is the first block to be accessed during the index scan methods, but at this stage, it is used to also store the actual index entries as well. After inserting rows into the table, the index block is filled with new entries. When the root index block is full, two new index blocks are then allocated, and index entries are moved to the new leaf blocks. The contents of the previously single filled block are replaced with pointers to the two new blocks. In this situation, the *BLEVEL* of the index is 1 and its *HEIGHT* is 2. An index increases in height whenever the index root block splits and the two new allocated blocks result in a new level within the index structure. However, the index root block remains the same throughout the entire life of the index [23] [27] [46].

B-tree indexing strategy follows robust improvement by accessing the data. Table 4.1 shows the reflection of the indexed data amount and depth of the index, which is calculated by *blevel* attribute of the *user_indexes* data dictionary. 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:

```
     SELECT power(2, i), blevel + 1 as depth
       FROM user_indexes
         WHERE index_name = 'STREAM_INDEX';
```

NUMBER OF RECORDS	INDEX DEPTH
1	1
2	1
256	1
512	2
1 024	2
262 144	2
524 288	3
1 048 576	3
134 217 728	3
268 435 456	4

Tab. 4.1: Index depth correlation

From the above table, it is evident, that the traversing using index provides a powerful power. Even if the table has 300 million of rows, just 4 index nodes must be accessed to reach the leaf layer consisting of the ROWID pointers. It means it will take four I/Os to find the key in the index 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 using an assumption, that significant data amount will be obtained and composed as part of the result set.

B-tree index structure is a robust solution, it does not degrade over time, ensuring the efficiency regarding the whole table size. However, there are three significant drawbacks – null value coverage, migrated row problem, and transaction support [49] [54].

First of all, the B-tree index cannot cover undefined tuples. They cannot be mathematically sorted. Thus, it is not possible to locate such representation inside the index. As a result, if the result set can contain such data, the index cannot be used at all, forcing the system to use sequential data scanning [36]. Let´s have a table consisting of the temperature data obtained by the sensor. For simplicity, let obtained values be correlated just with the validity:

```
    CREATE TABLE temperature_tab
                    (validity_date date PRIMARY KEY,
                     temperature_value number);
```

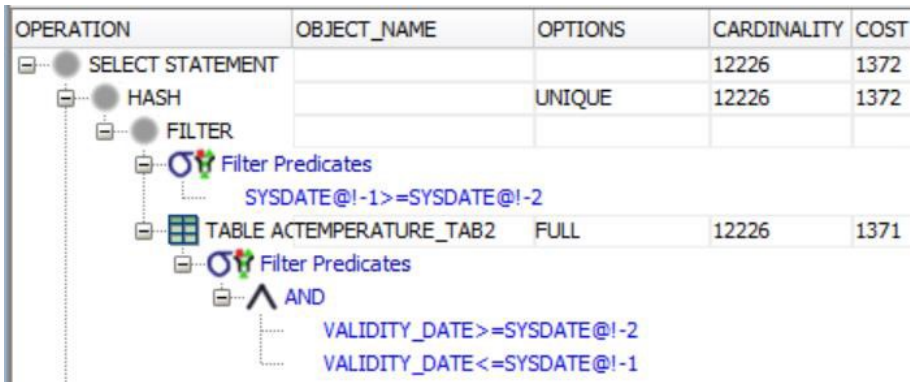
Regardless of the data amount, if we want to get values obtained by the specific time range, whereas undefined values can be present, sequential data scanning must always be used. Referencing the table structure, attribute *temperature_value* can hold a NULL value, whereas there is no limiting column constraint. Although such a situation does not occur in practice, the system cannot ensure it by the statistics, resulting in sequential data scanning.

```
  SELECT COUNT(*) as "Data amount",
         COUNT(temperature_value) as "Undefined temperature values",
          COUNT(distinct temperature_value) as
         "Unique values for temperature measurement"
    FROM temperature_tab;
```

DATA AMOUNT	UNDEFINED TEMPERATURE VALUES	UNIQUE VALUES FOR TEMPERATURE MEASUREMENT
1 000 000	0	300 000

Tab. 4.2: Data amount perspective

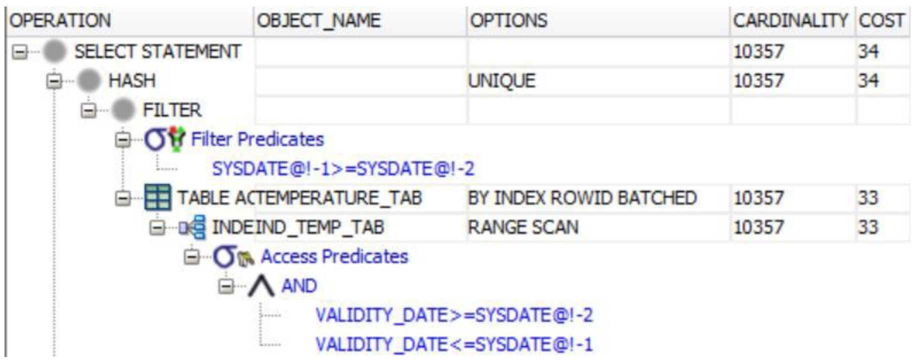
Table 4.2 shows the above query result. The aim is to get unique values for a particular day. Firstly, all data are loaded to the instance memory block-by-block by extracting the *validity_date*. If it falls into the time interval, it is covered by the consecutive step of hashing values to remove duplicates. If not, it is refused. The total costs for 1 million rows are 1 372. Figure 4.2 shows the execution plan. Whereas generally, undefined values can be present, a table is sequentially scanned using the Table Access Full method.



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			12226	1372
HASH		UNIQUE	12226	1372
FILTER				
Filter Predicates				
SYSDATE@!-1>=SYSDATE@!-2				
TABLE ACTEMPERATURE_TAB2		FULL	12226	1371
Filter Predicates				
AND				
VALIDITY_DATE>=SYSDATE@!-2				
VALIDITY_DATE<=SYSDATE@!-1				

Fig. 4.2: Execution plan handling NULLs – Table Access Full

A range scan can be used properly by removing NULL handling from the result set, as shown in Figure 4.3. It is ensured that all particular rows are index referenced.



OPERATION	OBJECT_NAME	OPTIONS	CARDINALITY	COST
SELECT STATEMENT			10357	34
HASH		UNIQUE	10357	34
FILTER				
Filter Predicates				
SYSDATE@!-1>=SYSDATE@!-2				
TABLE ACTEMPERATURE_TAB		BY INDEX ROWID BATCHED	10357	33
INDEIND_TEMP_TAB		RANGE SCAN	10357	33
Access Predicates				
AND				
VALIDITY_DATE>=SYSDATE@!-2				
VALIDITY_DATE<=SYSDATE@!-1				

Fig. 4.3: 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.

However, systems must cover undefined values due to the data failures, delays, and reliability aspects. 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 measurements. Although it can be partially solved by the undefined (NULL) value transformation [14], a particular reference is a function encapsulated. So, such a 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 [33] [39].

The second limitation is related to the physical database. Files are fixed-size block-shaped. Thus, fragmentation across the blocks can be present. Thus, 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, even if it still fits the block size, is stored in another block. Such a principle is absolutely correct and does not have a significant impact on the performance using the index. 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 [40] [49]. In general, it reflects less than 1% of additional demands, reflected by the I/O block loading necessity. The remaining free space in the blocks is used for the update operations. 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. From the index access point of view, both solutions require additional data blocks to be handled. Namely, for overflow block, by data accessing, multiple blocks must be loaded, despite the fact, that 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 always need to 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 Figure 4.4 holding the 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 [7] [55].

In [VIII], 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 [49].

Figure 4.4 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. The 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 the 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 to new).
5. Applying the change using the temporary structure – access path.
6. Changing the original ROWID for the index.

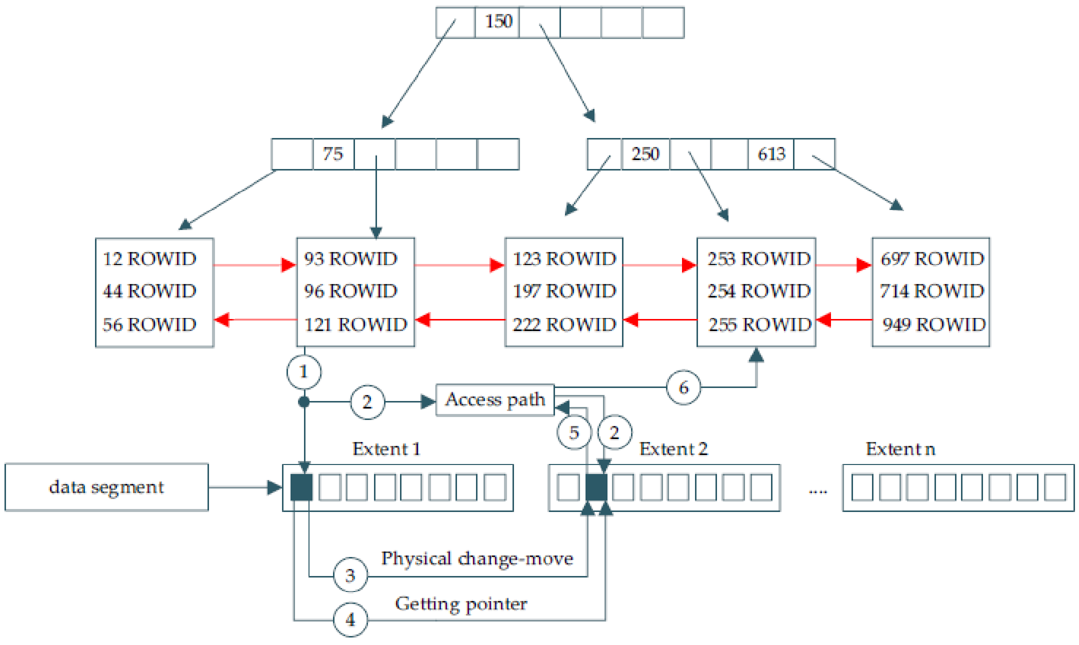


Fig. 4.4: B+ tree index highlighting the migrated row problem

As stated in [14], 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. 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 [14] [41].

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, the transaction definition is extended to cover the index set complexly. That can be, however, the bottleneck of the whole system. The number of indexes must be, therefore, balanced to optimize data access by the retrieval, but the other operations, which 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 [61].

When dealing with the B-tree indexes, the defined limitation can be even more significant due to the index balancing necessity at any time. 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.

4.1.1 INDEX-ORGANIZED TABLE

Data in the index-organized table (IOT) are stored in a B-tree index structure sorted according to the primary key. This is different from an ordinary table in which data is stored in an unordered way. Thus, the IOT structure is suitable when most of the column values of the index are included in the primary key. Creation of the IOT in Oracle is provided by adding the *ORGANIZATION INDEX* statement to the end of the table creation statement as follows:

```
     CREATE TABLE person
                  (id number,
                   name varchar2(50),
                   CONSTRAINT person_pk PRIMARY KEY(id, name))
       ORGANIZATION INDEX;
```

There is no need to access a row in the database from an index structure, so the total number of I/O operations needed to retrieve data can be reduced. The overall space required for the table is also reduced because there is no need to link to a row in a table, so the ROWID is not stored in the index.

Unlike Oracle, other databases use the term *clustered index* for the concept of the IOT. For example, in MySQL, the clustered index is created automatically when a table with a primary key or unique key is created. It is a special index because it is stored together with the data in the same table [74].

4.1.2 UNIQUE INDEX

The complementary feature of a B-tree index is its ability to be unique. There is a guarantee that any inserted values into the corresponding table will not be the same, so this index can be used to enforce the uniqueness of the data. An error occurs when there is an attempt to add a row with the key value matching an existing row. A unique index cannot be created on a single column containing the NULL value in more than one row. Similarly, it cannot be created on multiple columns if the combination of columns contains the NULL value in more than one row because they would be treated as duplicate values for indexing purposes [35]. To define an index to be unique, the following statement is used:

```
     CREATE UNIQUE INDEX ind_customer
       ON customer(surname, name);
```

4.1.3 REVERSE KEY INDEX

This type of index reverses the bytes of the key-value to solve the problem of block contention on leaves on the right side of the b-tree index structure which can be caused when primary keys are generated by a sequence. It means, it is useful for balancing I/O in an index with many sequential inserts which would have many similar values clustered together. For example, when two primary keys are 1234 and 1235, a reverse key index changes them into 4321 and 5321 and after that adds them into an index. These new primary keys are not stored next to each other but are spread in different blocks of the structure. A reverse key index is created by including the *REVERSE* clause as follows:

```
CREATE INDEX ind_subject
  ON subject(id) REVERSE;
```

It is not possible to create a bitmap index or an index-organized table as the reverse. The big disadvantage of a reverse key index is that an index range scan method cannot be used. Data are not sorted as it would be needed for this method because they are distributed over all the leaf nodes of the tree structure.

4.1.4 COMPOSITE INDEX

An index that contains multiple columns is known as a composite, concatenated, combined, or multi-column index. This type of index can speed the retrieval of data for the *SELECT* statements in which there are all or the leading portion of the columns of the index in the *WHERE* clause. Thus, the column order in the index is very important. In general, the most commonly accessed columns should be the first columns of the index. A composite index can be created as follows:

```
CREATE INDEX ind_vehicle
  ON vehicle(brand, model);
```

Compared to a number of single-column indexes, one composite index is more suitable because it saves storage space and saves the maintenance overhead for multiple single indexes [41].

4.1.5 KEY COMPRESSED INDEX

For reducing the storage and I/O requirements of concatenated indexes where the leading column is often repeated, it is useful to use key-compressed indexes. The term *COMPRESS [N]* indicated compression is required, where N specifies the prefix or a number of keys to compress. The default for index creation is *NOCOMPRESS*. To create this type of index the following statement is used:

```
CREATE INDEX ind_customer2

  ON customer(surname, name, id) COMPRESS 2;
```

The compression of the index can be enabled or disabled via rebuilding the index as follows:

```
ALTER INDEX ind_customer2 REBUILD NOCOMPRESS;
```

Key compression breaks an index into a prefix and a suffix entry. Compression is obtained by sharing the prefix entries among all the suffix entries in an index block, which can bring enormous savings in space. Subsequently, it is possible to store more keys for each index block while improving performance [45] [69].

Key compression is useful in situations with a unique multicolumn index or with a non-unique index where ROWID is appended to make the key unique. In the situation with a non-unique index, the duplicate key is stored as a prefix entry on the index block without the ROWID. The remaining rows become suffix entries consisting of only the ROWID.

It is not possible to create a key-compressed index on a bitmap index.

4.1.6 DESCENDING INDEX

Indexes are stored in ascending order by default. That is, the numbers are arranged from the smallest to the largest values, and the characters and strings as well. If there are some queries sorting some columns in ascending order and other columns in descending order, it is useful to define an index to be descending by using a keyword *DESC* as follows:

```
CREATE INDEX ind_person

  ON person(id, name DESC);
```

A descending index is not used on the bitmap index and the reverse-key index [35].

4.2 BITMAP INDEX

A bitmap index is a specific type of 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, which means a small level of uniqueness of column values, with a high number of table tuples. Thus, it is used mostly for columns, which have a significant number of duplicate values. In comparison with other indexes, dynamic storage requirement reduction can be identified. With a small number of CPUs, dramatic performance gains are present, due to binary operation processing. 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 [42].

Table 4.3 shows a table with a column of genders, in which there are only 2 unique values for males and females.

ID	NAME	GENDER
1	John	M
2	Alice	F
3	Mike	M
4	Amy	F
5	Kate	F
6	Smith	M
7	Peter	M

Tab. 4.3: A table with a suitable column for bitmap index

Bitmap indexes store information in bit arrays known as bitmaps. They are two-dimensional arrays with one column for every row in the table being indexed. A bitmap is stored for each index key and each index key stores pointers to multiple rows. Oracle uses a mapping function for converting each bit in the bitmap to the corresponding ROWID of the table. A bitmap index can be created as follows:

```
CREATE BITMAP INDEX ind_bitmap

  ON person(gender);
```

Bitmapped indexing is very beneficial in situations when one table includes multiple bitmap indexes. The creation of multiple bitmap indexes provides a powerful method for rapidly answering difficult SQL queries.

Figure 4.5 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. For the data retrieval, individual rows are processed by the binary operation merging.

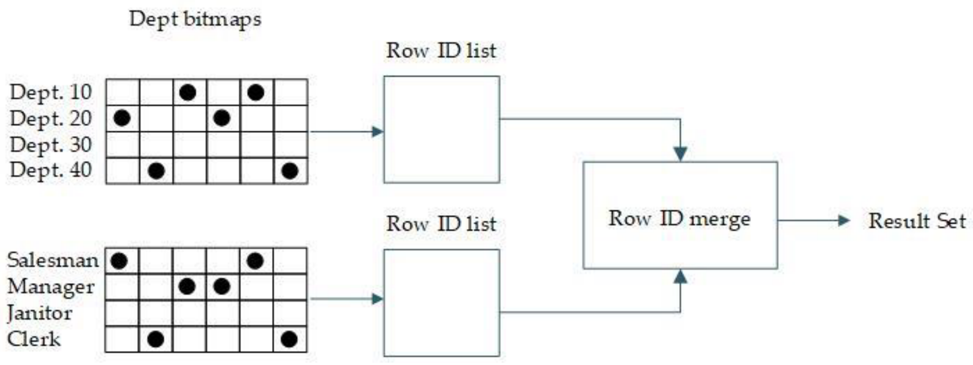


Fig. 4.5: Architecture of Bitmap index

Bitmap indexes are suitable for SQL statements with many *AND*, and *OR* operators in the *WHERE* clause. This index cannot be unique. It is used in systems for reporting and data analysis, known as data warehouse environments. These systems pull data from various sources together [58]. It is not recommended to use a bitmap index in *Online Transactional Processing Databases* which typically involve many operations of insert, update, and delete.

In the example below, there is a table with two bitmap indexes on the columns with a vehicle type with 3 possible values and an indication of whether the vehicle has been crashed with two possible values.

ID	TYPE	BOUGHT	CRASHED
1	Car	01.02.2020	N
2	Bicycle	23.02.2020	N
3	Motorbike	24.02.2020	Y
4	Car	01.03.2020	Y

Tab. 4.4: A table with two bitmap indexes

Each value of the column with a bitmap index has its own bitmap which is shown in Table 4.5. Each digit represents a single row of the table, where *1* means that the value occurs in the given row, and *0* means that it does not occur.

TYPE	BITMAP
Bicycle	0100
Motorbike	0010
Car	1001

CRASHED	BITMAP
Y	0011
N	1100

Tab. 4.5: Bitmaps

The following query is used to select vehicles meeting the criteria for both bitmap index columns:

```
SELECT *
  FROM vehicle
     WHERE type = 'car' AND crashed = 'Y';
```

The bitmap index of both the columns will be searched and logical operation AND will be performed on those bits:

```
1001   (a car)
AND
0011   (crashed)
----
0001   (result: a crashed car)
```

According to the resulting bitmap with the value 0001, it can be concluded that the searched value is located in the fourth row.

4.3 FUNCTION-BASED INDEX

This type of index is used for columns having the function applied to them, which need to be deterministic, for example:

```
CREATE INDEX ind_employee
   ON employee(LOWER(name));
```

The definition of function in the index must be exactly the same as in the *WHERE* clause. If it differs, the index is not used [66]. The function-based index can be bit-map, B-tree, or unique.

4.4 PREFIX INDEX

To save disk space on an index file, and to speed up insert and search operations, MySQL uses prefix indexes. They ensure that the entire length of the table column does not have to be used in the index, but only the number of characters defined for the column will be used for searching. For example, string columns are usually defined with a length of 255, but only a small part of them is used. In the following example, an index uses only the first 5 characters of the name column:

```
CREATE INDEX ind_prefix

  ON person(name(5));
```

4.5 VIRTUAL INDEX

It is possible to create an index which has no physical segment by using *NOSEGMENT* clause as follows:

```
CREATE INDEX ind_person2

  ON person(name) NOSEGMENT;
```

The nosegment indexes are used according to the value of the parameter *_use_nosegment_indexes*, which can be set as follows:

```
ALTER SESSION SET "_use_nosegment_indexes"=true;
```

When working with a large index without wanting to allocate space and consume resources, with an intention to only determine if the index would be used by the optimizer, it is useful to use a virtual index. When creating an index with the *NOSEGMENT* keyword, it is possible to test the scenario. After determining that the index would be useful, it is possible to drop the index and replace it without using the *NOSEGMENT* clause [35].

4.6 INVISIBLE INDEX

Marking indexes as invisible make them be ignored by the optimizer. However, they are maintained like any other indexes. They are useful in situations when testing how dropping an index would affect performance. Indexes are becoming invisible using the keyword *INVISIBLE* as follows:

```
CREATE INDEX ind_employee2

  ON employee(name) INVISIBLE;
```

Indexes can be altered to change the visibility as follows:

```
ALTER INDEX ind_employee2 VISIBLE;

ALTER INDEX ind_employee2 INVISIBLE;
```

4.7 HASH INDEX

A hash index stores collections of buckets organized in an array. It uses a hash function mapping the values of index keys to corresponding buckets stored in the hash index. Each bucket stores a list of pointers to the individual rows which belong to it. It 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 should distribute data as uniformly as possible [41]. Figure 4.6 shows different index keys that are mapped to different buckets in the hash index. F(x) is the name of the hash function.

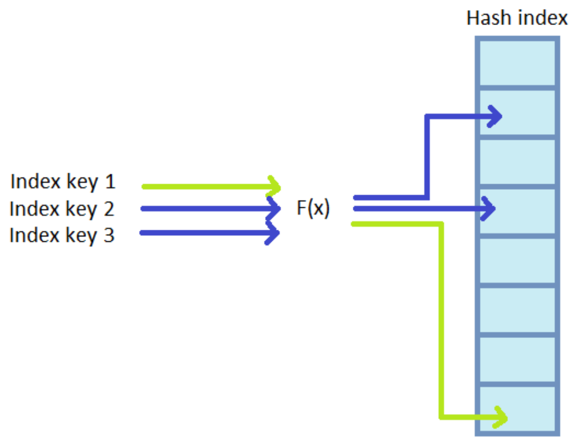


Fig. 4.6: Mapping of a hash function

Figure 4.7 shows the principle of using hash index during the data evaluation. Obtained data are shifted to the hast function as an input resulting in assigning a 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 [42].

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 [72]. 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.

The second problem is the dimensioning of the field structure itself in size. Data should be uniformly distributed across the buckets, covering the 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. 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 [13].

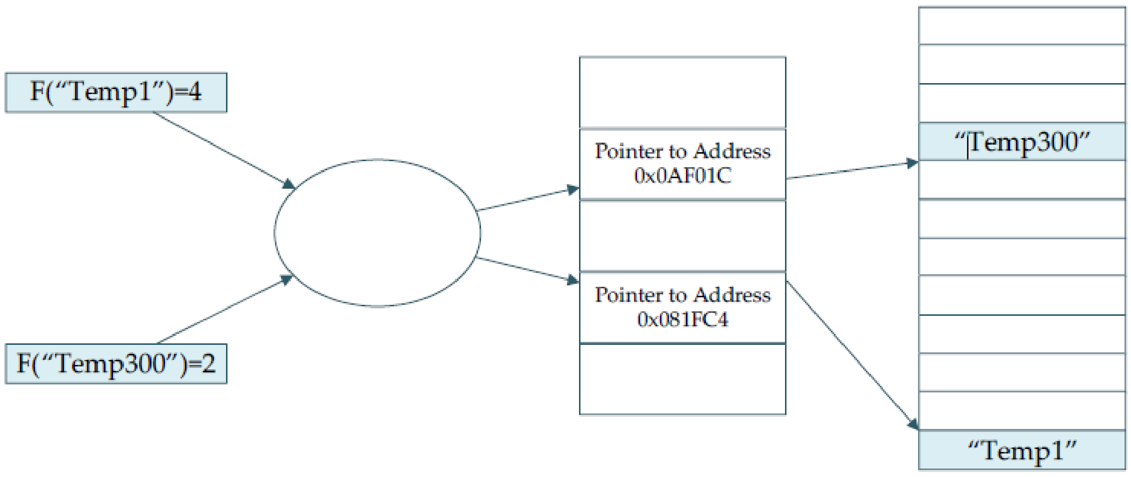


Fig. 4.7: Hash index

4.8 PARALLEL AND NOLOGGING CLAUSE

Creating an index offers 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 indexes 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 fully scanned, up to the last associated block (delimited by the *High Water Mark* [41]). 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 [70]. The following code highlights the description of parallelism. Based on [6], 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.

```
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 [21]. No matter how the data are processed, any change at the index level must be covered by the transaction. 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 in 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. 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 us to skip the transaction logging process related to the index [41]. 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 [51]. 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.

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 [68].

4.9 AUTOMATIC INDEXING

Automatic indexing was proposed in February 2019 and is now available just for the DBS Oracle, removing the database administrator intervention necessity for index management and structure optimization. It automatically creates, rebuilds, and drops indexes. The execution of these operations is based on the application workload. It analyzes the current workload and identifies candidates for indexes and validates them before implementation. Indexes are created as invisible, and it is tested whether they improve the performance. If so, then they can be altered to be visible and used in the application. However, if the performance is not improved, indexes are marked as unusable and subsequently are dropped. 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. In addition to the importance of sophisticated automation, transparency is also important, so all tuning activities are auditable via reporting [66].

Auto-created indexes are marked by the *SYS_AI* name denotation as a prefix. 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 responsible for the implementation itself.

```
dbms_auto_index.configure('AUTO_INDEX_MODE',
                          {'IMPLEMENT' | 'REPORT_ONLY'});
```

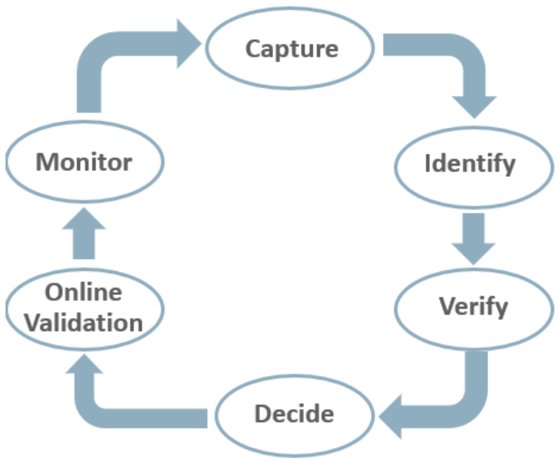


Fig. 4.8: Automatic Indexing Methodology

The automatic indexing methodology is based on a common approach to manual SQL tuning. It periodically **captures** the application´s SQL history into a SQL repository including plans, bind values, execution statistics, etc. Consequently, candidate indexes that may benefit from the newly captured SQL statements are **identified** and are created as unusable and invisible. Indexes that are obsoleted by newly created indexes are then dropped. The optimizer **verifies** if index candidates will be used for captured SQL statements. The indexes are materialized, and SQL runs to validate that the indexes improve their performance. The verification is done outside the application workflow. The next step is to **decide** on the type of index. The indexes are marked as visible if performance is better for all statements. The indexes remain visible if performance is worse for all statements. If performance is worse only for some statements, the indexes are marked visible except for the SQL statements that are regressed. The **validation** of the new indexes continues for other statements online. However, not more than one session executing a SQL statement can use new indexes. After that, index usage is continuously **monitored**. Indexes that have been created automatically and have not been used in a long time are dropped [37].

Automatic indexing can be applied to both tuned and un-tuned applications. In tuned applications, existing secondary indexes may be outdated, or important ones can be missing. Some secondary indexes can be dropped, and auto indexes can be added. As for un-tuned applications, existing indexes support primary or unique key constraints. Automatic indexing is applicable to all stages of an application lifecycle and can be fully manageable if desired.

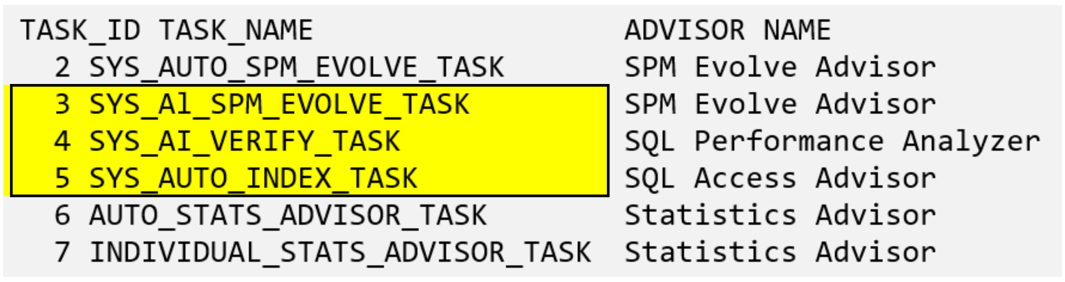
There is no performance regression done by the automatic tasks – analyzers and advisors. Figure 4.9 shows the list of advisors supervising the auto-indexing process.

```
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_Al_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
```

Fig. 4.9: Auto-indexing advisor list

For the automatic index configuration, a separate index tablespace can be created. Based on the [49], 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.

```
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 on 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 [41].

4.10 INDEX SCAN METHODS

There are several types of access paths used by the query optimizer to produce the best execution plan. An index scan retrieves data from an index based on the value of one or more columns in the index. The indexed column values accessed by the statement are searched in the index. If only columns of the index are accessed by the statement, the indexed column values are read directly from the index, rather than from the table.

The type of data that is retrieved by the index varies in different databases. In Oracle, the index search results in the values of the indexed attributes and the record address called ROWID. The result does not have to be always a ROWID, such as in MySQL. In MySQL, the result is a clustered or a secondary index which refers to a primary index formed by a primary key. The clustered index is essentially the table. This concept is called an index-organized table in Oracle [9] [71].

4.10.1 INDEX RANGE SCAN

Using an index range scan brings to access to selective data. The range can be bounded on both sides or unbounded on one or both sides. Data is returned in the ascending order of index columns. Multiple rows with identical values are sorted in ascending order by ROWID. The optimizer chooses an index range scan in the following situations:

- One or more leading columns of an index are specified in conditions.
- 0, 1, or more values are possible for an index key.

4.10.2 INDEX UNIQUE SCAN

A scan returning at most a single ROWID is an index unique scan. It is performed if a statement contains a *UNIQUE* or a *PRIMARY KEY* constraint so that it can guarantee that only a single row is accessed. This scan searches the index in order for the specified key. It stops processing as soon as it finds the first record because no second record is possible. The database obtains the ROWID from the index entry and then retrieves the row specified by the ROWID.

4.10.3 FULL INDEX SCAN

The entire index is scanned in order by a *Full Index Scan*. It reads the blocks singly. The optimizer chooses an index range scan in the following situations:

- A predicate references a column in the index. This column need not be the leading column.
- No predicate is specified and all columns in the table and in the query are in the index.
- No predicate is specified and at least one indexed column is not null.
- A query includes an *ORDER BY* on indexed non-nullable columns.

4.10.4 FAST FULL INDEX SCAN

This scan reads the index blocks in unsorted order, as they exist on disk. It does not use the index to probe the table, but reads the index instead of the table, essentially using the index itself as a table. The optimizer considers a fast full index scan when a query only accesses attributes in the index. Unlike a full scan, a *Fast Full Scan* cannot eliminate the sort operation because it does not read the index in order. The database uses multiblock I/O to read the root block and all of the leaf and branch blocks. The database ignores the branch and root blocks and reads the index entries on the leaf blocks [50].

4.10.5 INDEX SKIP SCAN

A composite index is split into smaller subindexes using a skip scan method. In *Skip Scanning*, the initial column of the composite index is not specified in the query, so it means it is skipped. Skip scanning is advantageous if there are few distinct values in the leading column of the composite index and many distinct values in the nonleading key of the index.

4.11 JOIN STRATEGIES

Join operations are important for database management. When executing queries involving multiple tables, the query optimizer can choose from three strategies for performing join operation [16]:

- merge join,
- nested-loop join,
- hash join.

In terms of indexes, it is required to firstly process join conditions and conditions in *WHERE* clause before processing attributes from the *SELECT* statement.

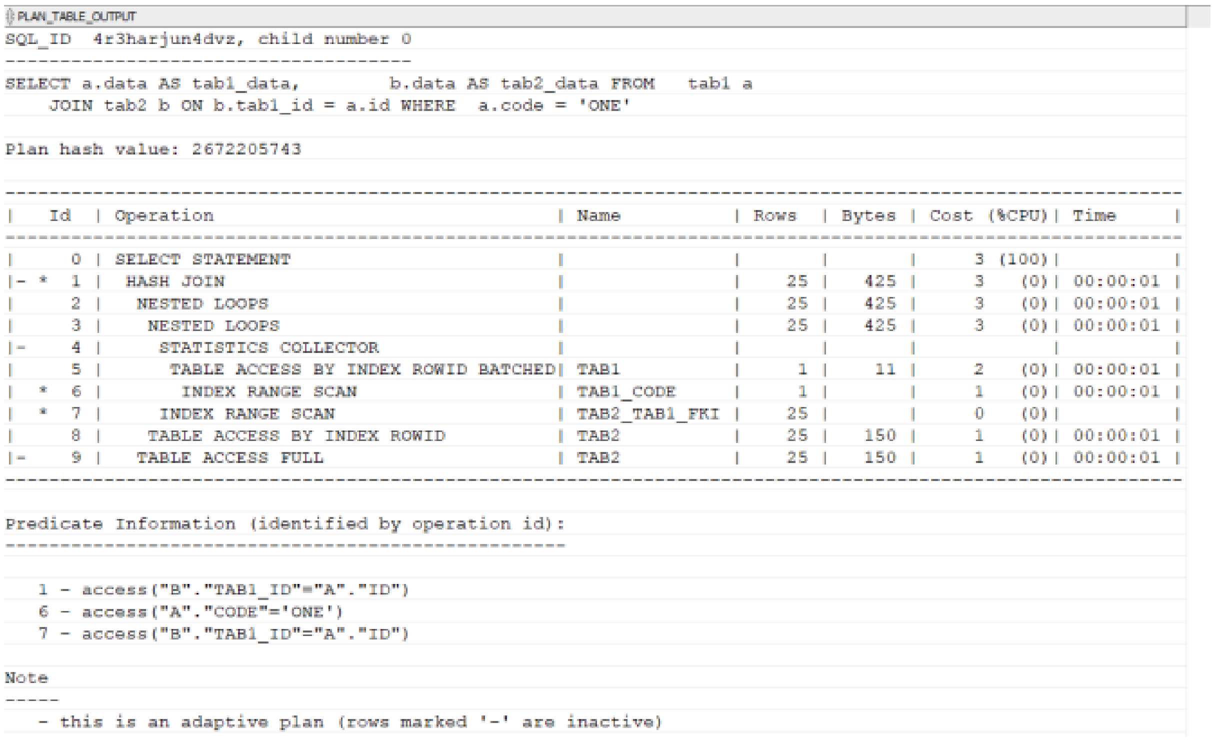
The *merge join* strategy is working with an index created on foreign keys and is considered to be the best strategy. The *nested-loop join* uses two nested loops. It means, it reads the rows from the first table in a loop and passes each row to a nested loop processing the second table in the join operation. It is the most demanding JOIN operation. The Nested Loop method limitation is associated with the sorting necessity in the first place, which can be time-consuming for large tables. During *hash join*, the optimizer uses the smaller of the joined tables and creates a hash table on the join key. It uses a hash function to specify the location in the hash table. This function divides data into small buckets and then treats 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 [41].

There exists also a *dynamic join*, which is a special type of join operation. In this type, two or more fields from two data sources are joined using a join condition that changes dynamically. The condition is based on the statistics [3].

Nested loop and Hash join methods strictly depend on the quality and accuracy of the statistics. The optimizer must select the appropriate method 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. Two server parameters secure definition and management. The original form of the *Optimizer_Adaptive_Features* has been marked obsolete in Oracle 12.2 [41].

```
ALTER {SESSION | SYSTEM}
  SET Optimizer_Adaptive_Features = {true | false}
    [scope = both];
```

Figure 4.10 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 Join, 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)
```

Fig.4.10: Dynamic execution plan of the query – table joining

4.12 SHRINKING THE SPACE

Oracle has several commands to reclaim unused disk space for tables and indexes. As fewer blocks accessed are required, index and table scans can run faster. The following statement ensures reclaiming disk space and lowers the high-water mark:

```
ALTER TABLE <table_name> SHRINK SPACE;
```

Recovering the space without lowering the high-water mark is ensured with the keyword *COMPACT* as follows:

```
ALTER TABLE <table_name> SHRINK SPACE COMPACT;
```

Adding the keyword *CASCADE* ensures recovering space for the object and all dependent objects, but it does not alter the indexes.

```
ALTER TABLE <table_name> SHRINK SPACE CASCADE;
```

Tables with dependent function-based, domain, or bitmap join indexes cannot be shrunk. Shrinking an index is ensured by the following statement:

```
ALTER INDEX <index_name> SHRINK SPACE;
```

Shrinking an index compacts the index segment and the database will immediately release any space that has been freed up.

The disadvantage of shrinking is that the process is technically demanding in terms of transaction processing. In this situation, it is necessary to lock the blocks, which is more difficult in terms of parallel access to the index. Conversely, if the *COALESCE* command is used, the blocks are also freed but remain available for the index. This option is without using the locks and is less technically demanding [42].

4.13 FULL-TEXT SEARCH

A search comparing every word in a document, or a table is called a full-text search. It is used in word processors and text editors, in which you can find a word or phrase anywhere in the document and not only on an abstract or in a set of keywords. It is a more advanced way to search a database. Full-text search quickly finds all words in a table without having to scan rows and without having to know which column a term is stored in. It searches for words inside text data. It works by using *text indexes*. A text index stores positions for all terms found in the columns on which the index is created. Using a text index can be faster than using a regular index to find rows containing a given term. This technique differs from searching using predicates such as *LIKE*, *REGEXP*, and *SIMILAR TO*, because the matching is term-based, not pattern-based. String comparisons in full-text search use all the normal collation settings for the database. For example, if the database is configured to be case insensitive, then full text searches will be case insensitive [42].

In the Oracle database, a full-text technology called *Oracle Text* is used. It uses standard SQL to index, search, and analyze texts and documents. Oracle Text provides three types of indexes that cover all text search needs.

1. *Standard index type* – is suitable for traditional full-text retrieval over documents and web pages. The *CONTEXT* index type provides a rich set of text search capabilities and uses *CONTAINS* query operator. The index is created using the following statement:

```
CREATE INDEX ind_text_context

  ON book(title)

    INDEXTYPE IS CTXSYS.CONTEXT;
```

   Running the following statement retrieves the desired values:

```
SELECT title
  FROM book
    WHERE CONTAINS (title, 'sun') > 0;
```

2. *Catalog index type* – provides flexible searching and sorting at web speed as a *CTXCAT* index which uses *CATSEARCH* query operator. The *CTXCAT* index comprises sub-indexes, which are created using the following statements:

```
BEGIN
  CTX_DDL.CREATE_INDEX_SET('set_books');
  CTX_DDL.ADD_INDEX('set_books', 'publication_date');
  CTX_DDL.ADD_INDEX('set_books', 'page_count');
END;


CREATE INDEX ind_text_ctxcat
  ON book(title)
    INDEXTYPE IS CTXSYS.CTXCAT
      PARAMETERS ('index set set_books');
```

Running the following statement retrieves the desired values:

```
SELECT title
  FROM book
    WHERE CATSEARCH (title, 'autobiography',
                        'order by page_count desc') > 0;
```

3. *Classification index type* – is suitable for building classification or routing applications. The *CTXRULE* index uses the *MATCHES* query operator. The index is created using the following statement:

```
CREATE INDEX ind_text_ctxrule
   ON myquery(query)
      INDEXTYPE IS CTXSYS.CTXRULE;
```

Running the following statement retrieves the desired values:

```
SELECT classification
  FROM myquery
    WHERE MATCHES(query, 'Database Systems are at FRI') > 0;
```

5 PARTITIONING

A powerful technology that physically divides certain large objects of relational databases into smaller parts based on the logical division of the data is called *partitioning*. It allows tables, indexes, and index-organized tables to be subdivided into smaller pieces, enabling these database objects to be managed and accessed at a finer level of granularity. A partitioned part of a database object is called *a partition.*

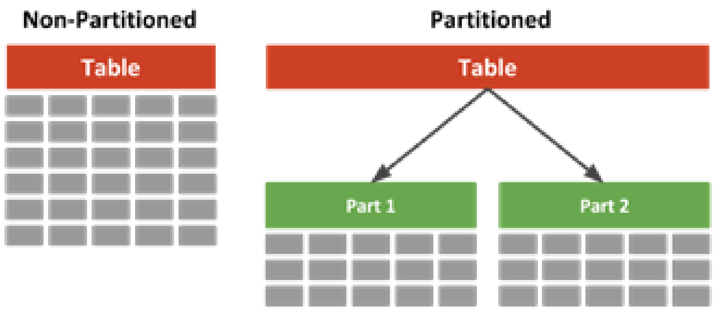


Fig. 5.1: A partitioned table

Each partition of a partitioned table can be partitioned again and so subpartitions are created.

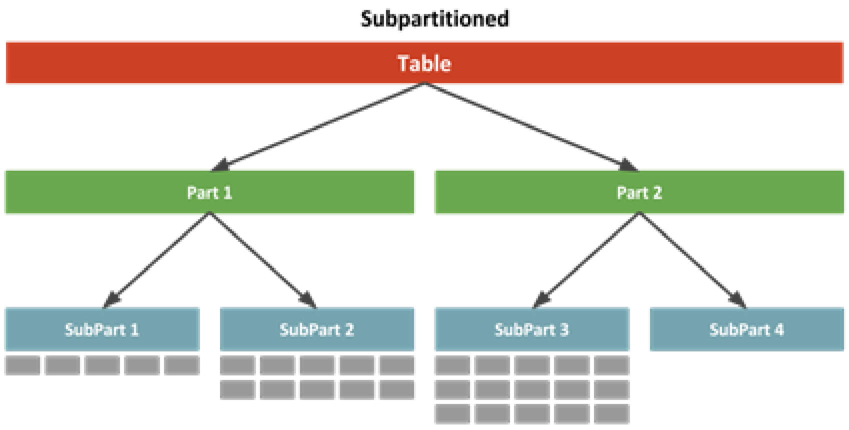


Fig. 5.2: A subpartitioned table

Database partitioning is normally done for manageability, performance, or availability reasons, or for load balancing.

- *Increasing performance* is provided by working only on the data that is relevant.
- *Improving availability* is performed on the basis of individual partition manageability.
- *Decreasing costs* is done by storing data in the most appropriate manner.

Partitioning is also easy to implement because it requires no changes to applications and queries. However, partitioning has worked on a different principle in the past. Instead of partitions created in one table, multiple tables were used. These tables had to have the same structure and therefore be *UNION* compatible. Based on this, it was subsequently possible to combine the results of the tables using the *UNION* operator [11].

5.1 PARTITIONING TECHNIQUES

There are three techniques by which partitions can be created. They are based on information allocation processes, which are *Hash*, *Range,* and *List*. According to them, there exist the following partitioning techniques:

- Hash Partitioning
- Range Partitioning
- List Partitioning

When creating partitions, each row in a partitioned table must be assigned only to a single partition. The data is divided into partitions based on a value called *a partition key.* This key can consist of one or more columns, which varies according to different partitioning techniques, which is shown in Table 5.1 [41].

	SINGLE-COLUMN	**MULTI-COLUMN**
HASH partitioning	✓	✓
RANGE partitioning	✓	✓
LIST partitioning	✓	X

Tab. 5.1: Availability for a single- and a multi-column partition key

Although list partitioning does not support a multi-column partition key, this limitation can be canceled by merging the values of multiple columns into a single column to be used as the partition key. This column will be populated automatically with the selected values, for example using a trigger.

It is important to select a partition key to be a column that is almost always used as a filter in queries. When it is so, it is not necessary to access all the data but only the relevant partitions. It means, *partition elimination* is used, and it can greatly improve performance when querying large tables.

5.1.1 HASH PARTITIONING

Using the hash partitioning technique, data are mapped to partitions based on a hashing algorithm that is applied to the identified partition key. The hashing algorithm distributes rows evenly among partitions, giving partitions approximately the same size. Hash partitioning is the ideal method for distributing data evenly across devices. It is also an easy-to-use alternative to range partitioning. Hash partitioning is suitable in situations, in which it is not possible to specify a range or define a list of values according to which the partitions are to be determined. For example, that can be identification numbers of products, employees, etc [78]. The following statement ensures the creation of a hash-based partitioning by hashing the product ID value. The number of partitions is specified there.

```
CREATE TABLE product (id INTEGER NOT NULL,
                      name VARCHAR2(60))
  PARTITION BY HASH (id)
    PARTITIONS 4;
```

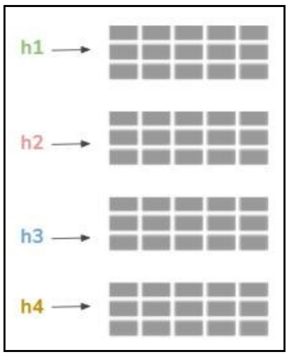


Fig. 5.3: Example of a hash partitioning strategy

5.1.2 RANGE PARTITIONING

The range partitioning technique maps data to partitions based on ranges of defined partition key values. Ranges must be specified for each partition so that it can contain rows for which the partitioning expression value lies within a given range. Ranges should be contiguous, but they cannot overlap with each other. In general, range partitioning is useful in cases when data can be logically segregated by some values. A classic example is time-based data. This partitioning technique is the most common one among the partitioning techniques. It is able to take advantage of partition elimination in many cases, including the use of exact equality and ranges - *less than*, *greater than*, *between*, and so on [67]. It offers the possibility to use a *MAXVALUE*, which represents a value that is always greater than the largest possible value. It serves as the least upper bound so it can catch all values that exceed the specified ranges.

The following statement ensures the creation of a range-based partitioning of sales based on their sale date.

```
CREATE TABLE sale (product_id INTEGER,
                   sale_date DATE)
  PARTITION BY RANGE (sale_date) (
       PARTITION sale1
             VALUES LESS THAN  (TO_DATE('01.05.2020','DD.MM.YYYY')),
       PARTITION sale2
             VALUES LESS THAN (TO_DATE('01.09.2020','DD.MM.YYYY')),
       PARTITION sale3
             VALUES LESS THAN (MAXVALUE)
  );
```

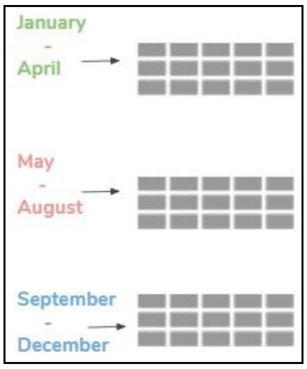


Fig. 5.4: Example of a range partitioning strategy

5.1.3 LIST PARTITIONING

In the list partitioning technique, there is a list of discrete values specified for each partition. It is usually preferred when there is only a limited set of partition key values. Unlike range and hash partitioning, multi-column partition keys are not supported for list partitioning. Only a single-column partition key can be used [78].

The following statement ensures the creation of list-based partitions for company offices, which are situated in the listed states of Europe, Asia, and Africa.

```
      CREATE TABLE company_office (id INTEGER,
                                   name VARCHAR2(20),
                                   state VARCHAR2(10))
        PARTITION BY LIST (state) (
          PARTITION europe
            VALUES ('Slovakia', 'Germany', 'Finland'),
          PARTITION asia
            VALUES ('India', 'China', 'Japan'),
          PARTITION africa
            VALUES ('Nigeria', 'Kenya', 'Ghana')
       );
```

A graphical view of the list partitioning strategy is shown in Figure 5.5.

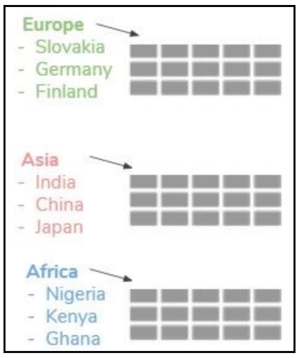


Fig. 5.5: Example of a list partitioning strategy

5.2 PARTITIONING METHODS

Database tables can be partitioned in two methods using the three above-described partitioning techniques. The partitioning methods are:

- Single-level Partitioning
- Composite Partitioning

5.2.1 SINGLE-LEVEL PARTITIONING

In single-level partitioning, only one of Hash, Range, or List data distribution methodology is used [78].

5.2.2 COMPOSITE PARTITIONING

This partitioning method is a combination of the basic partitioning techniques. It offers the benefits of partitioning on two dimensions. One data distribution method is used for a table to be partitioned and then each partition is further subdivided into smaller subpartitions using a second data distribution method.

Composite *hash-hash partitioning* enables two-dimensional hash partitioning. It is beneficial to enable partition-wise joining along two dimensions. Composite *hash-range partitioning* and *hash-list partitioning* firstly create partitions using a hash function which are sequentially subdivided into subpartitions based on a defined range or a list of values [65].

Composite *range-hash partitioning* creates partitions using the range method, where each partition is sequentially subdivided into subpartitions using the hash method. It provides the improvement of manageability of range partitioning and brings the data placement advantage of hash partitioning. Composite *range-range partitioning* enables two-dimensional range partitioning. For example, product data can be partitioned by their production date, and each partition can be then subpartitioned by its sale date. Composite *range-list partitioning* firstly uses the range method to create partitions, and within each partition, subpartitions are made using the list method [57].

Composite *list-hash partitioning* creates partitions using the list method, which are sequentially hash partitioned into subpartitions. Composite *list-range partitioning* enables using the range method on previously created list partitions. For example, dividing products into partitions based on the country of their production where each partition is subpartitioned using the date of production. Composite *list-list partitioning* enables two-dimensional list partitioning. For example, partitioning on the basis of the country and subsequently of the region [65].

5.3 PARTITIONING SCHEME

The mapping of logical partitions to physical groups of files is ensured by *a partitioning scheme*. These physical groups contain one or more data files that can be stored on one or more disks, as shown in Figure 5.6. It is possible to store all partitions in one group of files, but it is much more useful if each partition has its own group of files. Thus, data that is not accessed as often might be placed on slower disks, and those that require more frequent access might be stored on faster disks. What is more, filegroups can be backed up and restored individually. They can be also set to be read-only. For example, it is practical to set historical and unchanging data to be read-only, so they are excluded from regular backups [53].

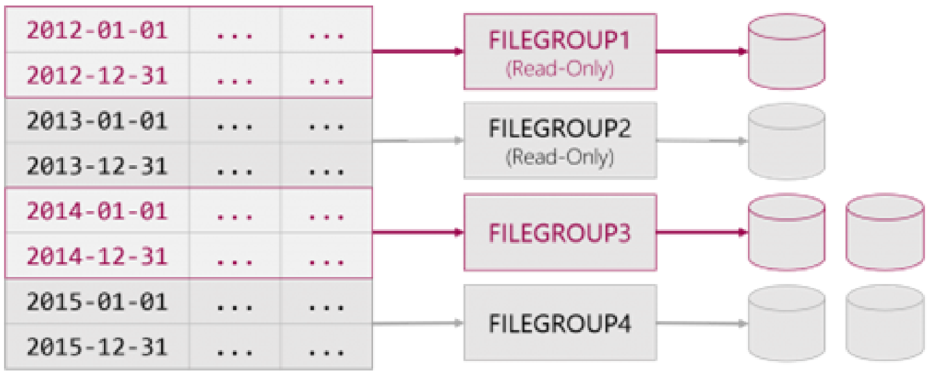


Fig. 5.6: Partition Scheme

5.4 PARTITIONED INDEX

The index can also be divided into partitions. It is possible to have partitioned tables without partitioned indexes, but also to have a non-partitioned table with partitioned indexes. Figure 5.7 shows a non-partitioned table with two types of indexes. The index on the left side is partitioned into three parts, and the index on the right side is non-partitioned.

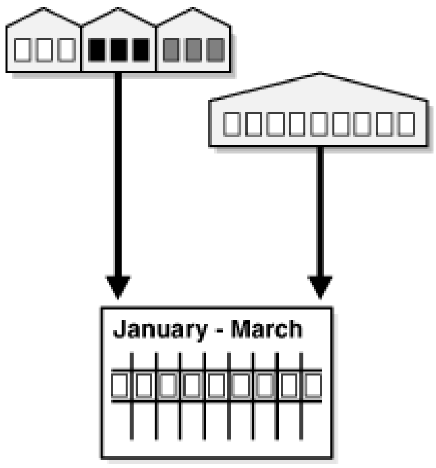


Fig. 5.7: Indexes of a non-partitioned table

A table partitioned into three partitions related to different months is shown in Figure 5.8. Two different indexes are linked to this table, the partitioned index on the left side, and the non-partitioned index on the right side [41].

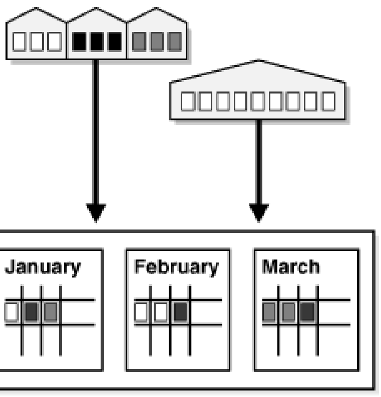


Fig. 5.8: Indexes of a partitioned table

If one partition of an index binds to one partition of a table, it is a local partitioned index. If one index partition binds to multiple table partitions, it is a global partitioned index.

A great availability is offered by a *local partitioned index*, which is also easier to manage than other types of partitioned indexes. When a partition of a table is changed, only one partition of the index needs to be changed. All of its keys refer only to rows stored in a single underlying table partition. It offers equipartitioning as each partition of a local index is associated with exactly one partition of the table. For this reason, the index partitions are automatically kept synchronized with the table partitions, so each table-index pair can become independent. It means they are both added, dropped, or split in the same way. Any actions making the data in one partition unavailable or invalid affect only a single partition. A new partition cannot be explicitly added to a local index. It can be added only when a new partition is added to the underlying table as well. Similarly, a partition cannot be explicitly dropped from a local index. It can be dropped only when a partition from the underlying table is dropped as well [66]. Figure 5.9 shows a structure of a local partitioned index and its relationship to the table partitions. The three top objects refer to index partitions and the bottom three objects refer to table partitions.

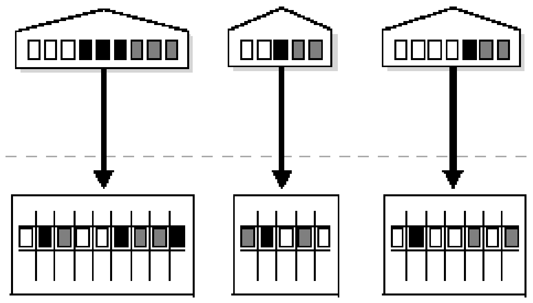


Fig. 5.9: Local partitioned index

A local index is created by specifying the *LOCAL* keyword as follows:

```
CREATE INDEX <index_name>
  ON <table_name>(<attribute1>[, <attribute2>, …] LOCAL;
```

A local partitioned index is *prefixed* if it is partitioned on a left prefix of the index columns. A local partitioning index that is not partitioned on a left prefix is referred to as a *non-prefixed index*. This type of index cannot be unique unless the index key is a subset of the partitioning key [43].

The partitioning key independent of the table's partitioning method is making *global partitioned indexes* more flexible. The fact that all rows in the underlying table can be represented in the index is ensured by a partition bound called *MAXVALUE*, which is higher than the highest partition of a global index. Figure 5.10 shows a structure of a global partitioned index and its relationship to the table partitions. The three top objects refer to index partitions and the bottom three objects refer to table partitions. Thus, unlike a local partitioned index, in this index, its partitions bind to different partitions of the table. *A global partitioned index* is much more difficult to manage than a local partitioned index because moving or removing data in one partition of a table can affect all partitions of the global index. Thus, the partitions are not independent [41].

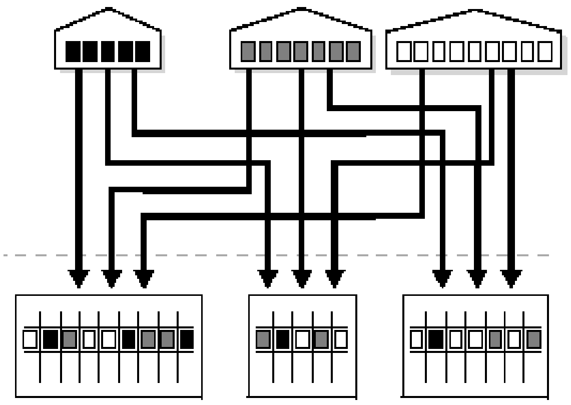


Fig. 5.10: Global partitioned index

A global index is created by specifying the *GLOBAL* keyword as follows:

```
CREATE INDEX <index_name>
  ON <table_name>(<attribute_name>) GLOBAL
    PARTITION BY RANGE (<attribute_name>) (
      PARTITION <partition1_name> VALUES LESS THAN (x),
      PARTITION <partition2_name> VALUES LESS THAN (y),
       …
      PARTITION <partitionN_name> VALUES LESS THAN (MAXVALUE)
);
```

5.5 PARTITIONING TO IMPROVE PERFORMANCE

Partitioning improves performance in various ways. *Partition pruning* works on the basis that unnecessary partitions are not processed in the query, and they are skipped. It means operations are performed only on those partitions that are relevant to the SQL statement. In the following example, a table was created with partitions by individual calendar months. Selecting data in the range according to the given condition does not require access to the whole table, but only access to two partitions belonging to January and February [50].

```
SELECT SUM(price)
  FROM sale
    WHERE sale_date BETWEEN to_date('01.01.2020', 'DD.MM.YYYY')
      AND to_date('13.02.2020', 'DD.MM.YYYY');
```

Improving the performance of multi-table joins can be also ensured by partitioning. A division of a large join into smaller joins between a pair of tables partitioned on the same partition key is executed by *a partition-wise join*. Smaller joins are created between each of the partitions.

6 AUTOMATIC STORAGE MANAGEMENT

Oracle Corporation provides a feature called *Automatic Storage Management* (ASM), which is a version of an autonomous cloud solution. Its aim is to simplify the management of database files, control files, and log files. It provides tools to manage file systems and volumes directly inside the database. Database administrators are allowed to control volumes and disks using SQL statements. They can reference disk groups that ASM manages instead of individual disks and files.

The main components of ASM are *disk groups*, each of which consists of several physical disks that are controlled as a single unit. The physical disks are called *ASM disks*, and the files in the disks are called *ASM files*. The content of files stored in a disk group is evenly distributed, or striped, so the uniform performance across the disks is provided. Disks can be added to a disk group or removed from it while a database continues to access files from the disk group. In this situation, ASM automatically redistributes the file contents and eliminates the need for downtime when redistributing the content. It also allows the addition of disks online. With ASM, manual load balancing is not necessary. When new disks are added to a disk group, rebalancing happens automatically without a disconnection.

To stripe data, ASM separates files into stripes and spreads data evenly across all of the disks in a disk group to balance loads and reduce I/O latency. Files are divided into equal-sized extents.

ASM also provides automatic mirroring of ASM files. The mirroring occurs at the extent level. If a disk group is mirrored, each extent has one or more mirrored copies, and mirrored copies are always kept on different disks in the disk group [75].

7 CURRENT RESEARCH PERSPECTIVES

This chapter contains a description of selected institutions and universities that deal with issues similar to the areas of our thesis. We discuss their research, which is mainly focused on indexing, partitioning, or join techniques.

7.1 MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Massachusetts Institute of Technology (*MIT*) is a private land-grant research university in Cambridge, Massachusetts. Established in 1861, MIT has since played a key role in the development of modern technology and science, ranking it among the top academic institutions in the world. Their research institute called Computer Science and Artificial Intelligence Laboratory established in 2003 deals with database systems. In [77] they have dealt with partitioning techniques for fine-grained indexing. Many data-intensive websites use databases that grow much faster than the rate that users access the data. Such growing datasets lead to ever-increasing space and performance overheads for maintaining and accessing indexes. There is often considerable skew with popular users and recent data accessed much more frequently. These observations led them to design Shinobi, a system that uses horizontal partitioning as a mechanism for improving query performance to cluster the physical data and increasing insert performance by only indexing data that is frequently accessed.

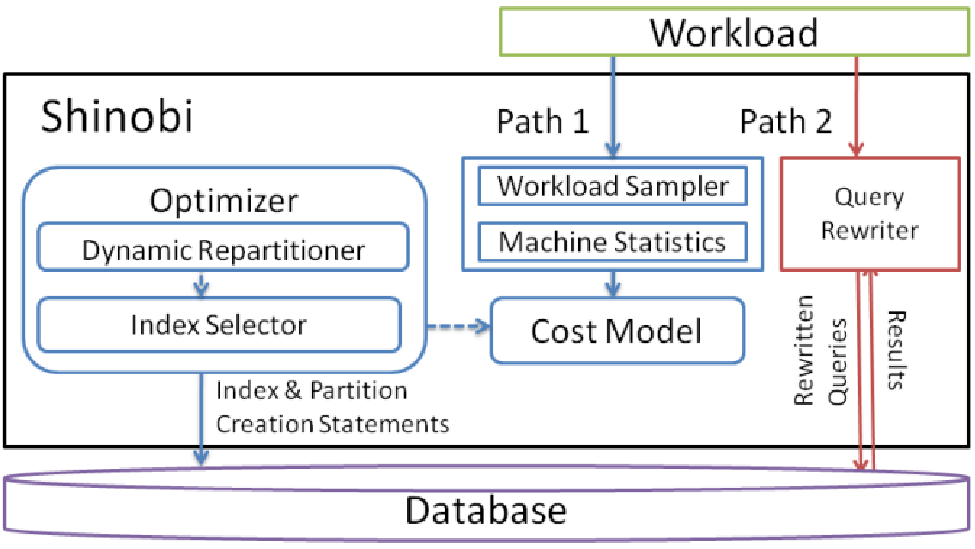


Fig. 7.1: The Shinobi architecture

They have presented database design algorithms that optimally partition tables, drop indexes from partitions that are infrequently queried, and maintain these partitions as workloads change. They have shown a 60x performance improvement over traditionally indexed tables using a real-world query workload derived from a traffic monitoring application. Their contribution toward partitioning in a single-machine database are as follows:

- *Enabling selective indexing with partitioning*. Shinobi chooses the optimal partitions to index, which dramatically reduces the amount of data that is indexed. In their experiments using a workload from Cartel, Shinobi can avoid indexing over 90% of the table and reduce index update costs by 30x as compared to a fully indexed table without sacrificing performance.
- *Partitioning-based clustering*. Shinobi optimally partitions tables for a given workload, which increases query performance by physically co-locating similarly queried data. Using the same Cartel workload, they improve query performance by more than 90x as compared to an unpartitioned, fully indexed, table.
- *Reducing index creation costs*. Shinobi only indexes partitions that are frequently accessed. By splitting the table into smaller partitions, the cost of creating an index on a single partition becomes cheaper, which lets the system make fine-grained optimizations.
- *Novel workload lifetime estimation*. Shinobi uses a novel online algorithm that uses past queries to estimate the number of queries the workload will continuously access in a given data region.

In [65] they have dealt with an adaptive partitioning scheme for ad-hoc and time-varying database analytics. A large number of techniques have been proposed to efficiently partition a dataset, often focusing on finding the best partitioning for a particular query workload. However, many modern analytic applications involve ad-hoc or exploratory analysis where users do not have a representative query workload. Furthermore, workloads change over time as businesses evolve or as analysts gain a better understanding of their data. Static workload-based data partitioning techniques are therefore not suitable for such settings. They have presented Amoeba, an adaptive distributed storage system for data skipping. It does not require an upfront query workload and adapts the data partitioning according to the queries posed by users over time. They have presented the data structures, partitioning algorithms, and efficient implementation on top of Apache Spark and HDFS. Their experimental results showed that the Amoeba storage system provides improved query performance for ad-hoc workloads, adapts to changes in the query workloads, and converges to a steady-state in case of recurring workloads. On a real-world workload, Amoeba reduces the total workload runtime by 1.8x compared to Spark with data partitioned and 3.4x compared to unmodified Spark. Query optimizers tend to push predicates down to scan operators and big data systems like Spark SQL allow custom predicate-based scan operators. Amoeba integrates as a predicate-based scan operator and the re-partitionings happening to change data layout are invisible to the user. Adaptive partitioning/indexing is extensively used in modern single-node in-memory column stores for achieving good performance. These techniques, called Cracking have been used to generate an adaptive index on a column based on incoming queries. Partial sideways cracking extends it to generate an adaptive index on multiple columns. Cracking happens on each query and maintains additional structures to create the index. The reason cracking cannot be applied to a distributed setting is because the cost of re-partitioning is very high. Each round of re-partitioning needs to be carefully planned to amortize the cost associated with it. Their approach is complementary to many other physical storage optimizations. For example, decomposed storage layouts (i.e., column-stores) are designed to avoid reading columns that are not accessed by a query. In contrast, partitioning schemes, including their, aim to avoid reading entire partitions of the dataset. Although their prototype does not use a decomposed storage model, there is nothing about their approach that cannot work in such a setting: individual columns or column groups could easily be separately partitioned and accessed in their approach. In summary, they have made the following major contributions:

- They have described a set of techniques to aggressively partition a dataset over several attributes and propose an algorithm to generate a robust initial partitioning tree. Their robust partitioning tree spreads the benefits of data partitioning across all attributes in the schema. It does not require an upfront query workload, and also handles data skew and correlations.
- They have described an algorithm to adaptively repartition the data based on the observed workload. They have presented a divide-and-conquer algorithm to efficiently pick the best repartitioning strategy, such that the expected benefit of repartitioning outweighs the expected cost.
- They have described an implementation of their system on top of the Hadoop Distributed File System (HDFS)1 and Spark. This storage system consists of:
  (i) upfront partitioning pipeline to load the dataset into Amoeba,

(ii) an adaptive query executor used to read data out of the system.

- They have presented a detailed evaluation of the Amoeba storage system on real and synthetic query workloads to demonstrate three key properties:
  (i) *robustness*: the system gives improved performance over ad-hoc queries right from the start,
  (ii) *adaptivity*: the system adapts to the changes in the query workload,
  (iii) *convergence*: the system approaches the ideal performance when a particular workload repeats over and over again.

7.2 TECHNICAL UNIVERSITY OF MUNICH

Technical University of Munich (*TUM*) is a public research university in Munich and is considered one of the most research-focused universities in Europe. Department of Database Systems deals with the indexing of highly dynamic hierarchical data [24]. Maintaining and querying hierarchical data in a relational database system is an important task in many business applications. This task is especially challenging when considering dynamic use cases with a high rate of complex, possibly skewed structural updates. Labeling schemes are widely considered the indexing technique of choice for hierarchical data, and many different schemes have been proposed. However, they cannot handle dynamic use cases well due to various problems, such as lack of query capabilities, insufficient complex update support, and vulnerability to skewed updates. They have proposed the dynamic Order Indexes, which offer competitive query performance, unprecedented update efficiency, and robustness for highly dynamic workloads. They can be viewed as a dynamic flavor of nested intervals labeling, using the concept of accumulation to maintain node levels while supporting even complex and skewed updates efficiently. Their indexes considerably outperform prior techniques when considering complex updates on subtrees, sibling ranges, and inner nodes.

They have proposed the concept of these Order Indexes and three specific data structures *AO-Tree, BO-Tree, and O-List.* The evaluation shows how carefully choosing a suitable back-link representation and block size can further optimize performance. The BO-Tree with varying block sizes yields a particularly attractive query/update tradeoff, making it a prime choice for indexing dynamic hierarchies.

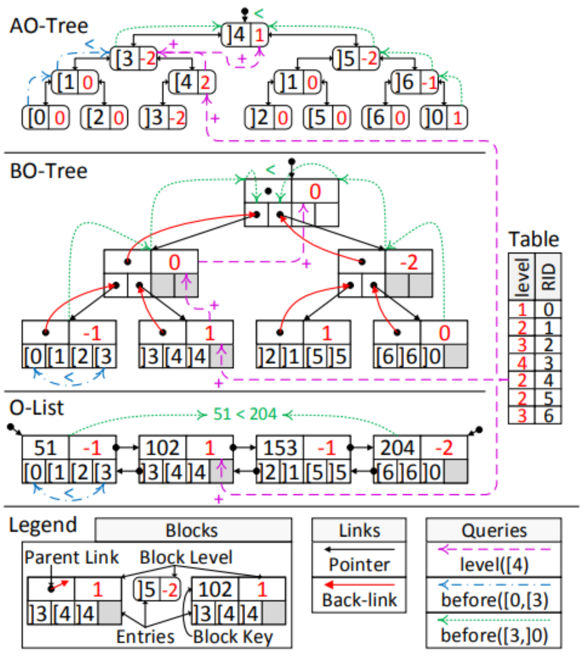


Fig. 7.2: Query evaluation in the Order Indexes

In [25] they have dealt with adopting worst-case optimal joins in relational database systems. Worst-case optimal join algorithms are attractive from a theoretical point of view, as they offer asymptotically better runtime than binary joins on certain types of queries. In particular, they avoid enumerating large intermediate results by processing multiple input relations in a single multiway join. However, existing implementations incur a sizable overhead in practice, primarily since they rely on suitable ordered index structures on their input. Systems that support worst-case optimal joins often focus on a specific problem domain, such as read-only graph analytic queries, where extensive precomputation allows them to mask these costs. They have presented a comprehensive implementation approach for worst-case optimal joins that is practical within general-purpose relational database management systems supporting both hybrid transactional and analytical workloads. The key component of our approach is a novel hash-based worst-case optimal join algorithm that relies only on data structures that can be built efficiently during query execution. Furthermore, they have implemented a hybrid query optimizer that intelligently and transparently combines both binary and multi-way joins within the same query plan. They have demonstrated that their approach far outperforms existing systems when worst-case optimal joins are beneficial while sacrificing no performance when they are not. Their implementation offers greatly improved runtime on complex analytical and graph pattern queries, where worst-case optimal joins have an asymptotic runtime advantage over binary joins. At the same time, it loses no performance on traditional OLAP workloads where worst-case optimal joins are rarely beneficial. They achieved this through a novel hybrid query optimizer that intelligently combines both binary and worst-case optimal joins within a single query plan and through a novel hash-based multi-way join algorithm that does not require any expensive precomputation. The contributions thereby allow mature relation database management systems to benefit from recent insights into worst-case optimal join algorithms, exploiting the best of both worlds.

7.3 UNIVERSITY OF ALBERTA

The University of Alberta is a public research university located in Edmonton, Alberta, Canada. The Department of Computing Science at the University of Alberta is the oldest and one of the largest computing science departments in Canada. Their Database Systems research includes database systems, spatiotemporal and image databases, information retrieval, content-based information retrieval, querying and indexing novel data types, indexing, data mining, etc. In [16] they have dealt with efficiently transforming tables for joinability. Data from different sources rarely conform to single formatting even if they describe the same set of entities, and this raises concerns when data from multiple sources must be joined or cross-referenced. Such a formatting mismatch is unavoidable when data is gathered from various public and third-party sources. Commercial database systems are not able to perform the join when there exist differences in data representation or formatting, and manual reformatting is both time-consuming and error-prone. They have studied the problem of efficiently joining textual data under the condition that the join columns are not formatted the same and cannot be equi-joined, but they become joinable under some transformations. The problem is challenging simply because the number of possible transformations explodes with both the length of the input and the number of rows, even if each transformation is formed using very few basic units. They have shown that an efficient algorithm can be developed based on the common characteristics of the joined columns and develop one such algorithm over a rich set of basic operations that can be composed to form transformations. The evaluation revealed that not only their algorithm covers a rich set of transformations but is also a few orders of magnitude faster than their competitor. Based on their analysis and experiments on benchmark datasets, the main limitations of their work can be broken down into the following three cases:

- There is a gap between the performance of a golden row matching and their row matching algorithm, and this gap varies for different datasets. This gap translates to noise that is passed to the transformation discovery, which can produce false transformations.

- They have limited the number of placeholders, which improved the running performance, but some transformations can be missed.
- Some tables cannot be joined using string-based transformations only. For example, their transformations cannot capture semantic relations such as when one value is a synonym of another value.

Based on a study of their contributions, they want to focus on expanding their approach in the future. It can be extended in a few directions. One direction is transfer learning where transformations obtained for one dataset can be adapted to another dataset. Extending their work to employ non-string-transformations considering the semantics of the words can be another interesting direction that deals with the limitations of textual transformation.

7.4 KARUNYA INSTITUTE OF TECHNOLOGY AND SCIENCES

Karunya Institute of Technology and Sciences is a private residential institute in Coimbatore, India. They have dealt with join techniques in [32]. Query processing in a relational database involves two parts: query execution and query optimization. Usually, query execution takes more time than query optimization. The most expensive part of query execution is joining. In the case of very large databases and highly normalized databases, the frequency of joining queries is high. They have compared the costs of Nested Loop Join, Hash Join, and Merge Sort Join. Join operations can be improved in two ways one in controlling the way in which tuples are stored e.g., the physical storage of data such as partitioning the table, composite index, join index, clustering of data tuples, etc. Another way is to improve the technology used to Join the relations. A small change to the technique improves the performance of Join dramatically. Using the optimizer, they have estimated the cost of each join method and chose the method with the least cost. A nested loop join is inefficient when a join returns a large number of rows (typically, more than 10,000 rows is considered large), and the optimizer might choose not to use it. When using the CBO, a hash join is the most efficient join when a join returns a large number of rows. When using the RBO, a merge join is the most efficient join when a join returns a large number or rows. Random record generation performs much better than the nested loop join. The number of iterations required to find the records satisfying the join conditions is 50% reduced in the Random record generation method. The hybrid hash join typically performs fewer I/O operations than the hash join. XJoin, based on the Symmetric Hash Join, occupies less memory than hash join. It extends the symmetric hash join to use less memory by allowing parts of the hash tables to be moved to secondary storage. It does this by partitioning its inputs, similar to the way that hybrid hash join solves the memory problems of classic hash join. MJoin is used to perform multi join. MJoin uses a single hash table for each input and is limited to supporting multi-way joins where all inputs are partitioned on the same attributes.

In [10] they have dealt with a term-based inverted index partitioning model for efficient distributed query processing. In a shared-nothing, distributed text retrieval system, queries are processed over an inverted index that is partitioned among a number of index servers. In practice, the index is either document-based or term-based partitioned. This choice is made depending on the properties of the underlying hardware infrastructure, query traffic distribution, and some performance and availability constraints. In query processing on retrieval systems that adopt a term-based index partitioning strategy, the high communication overhead due to the transfer of large amounts of data from the index servers forms a major performance bottleneck, deteriorating the scalability of the entire distributed retrieval system. In their work, to alleviate this problem, they have proposed a novel inverted index partitioning model that relies on hypergraph partitioning. In the proposed model, concurrently accessed index entries are assigned to the same index servers, based on the inverted index access patterns extracted from the past query logs. The model aims to minimize the communication overhead that will be incurred by future queries while maintaining the computational load balance among the index servers. They have evaluated the performance of the proposed model through extensive experiments using a real-life text collection and a search query sample. Their results showed that considerable performance gains can be achieved relative to the term-based index partitioning strategies previously proposed in the literature. Compared to document-based index partitioning, however, there is still a large performance gap that remains.

A possible extension to this work is a multiconstraint model, where the storage load imbalance can be captured as an additional constraint in the model. Another possible extension is to replace the adopted replication heuristic, which replicates frequently accessed inverted lists on all index servers, with the recently proposed techniques that couple hypergraph partitioning with replication. This may lead to a further reduction in communication costs.

8 OWN CONTRIBUTION

In this chapter, we discuss our own contributions in detail. The individual topics and experiments in the subchapters were also transformed into articles published at various international conferences.

8.1 MASTER INDEX

The limitation of using the *Full index scan* method category is based on the fact, that the context reflected by the *Where* clause conditions can be evaluated directly in the leaf layer of the index. Thus, although the order of the attributes inside the index does not fit the query, relevant attributes are present, however in non-suitable order. As a consequence, if the *ROWID* on the leaf layer is selected by passing the *Where* condition of the query, it is certain, that the record will contain the data needed to create the result set. Thus, no irrelevant data block is loaded into the memory *Buffer cache*, except for the migrated row problem [5] [8].

Our solution uses a different principle. If accepted, the defined index will be used in any case. If the index based on the attributes is not suitable for the query, it will be used only as the access path to the data blocks with the real data. The importance of our solution definition is described in the following example. Let´s have four data rows for the table. For simplicity, let´s assume, that each data row is located in a separate data block at the beginning. Then, insert two new rows, which will be located in the same data block. The blocks are associated with the object in the form of individual extents, not the blocks directly. So, let´s assume, that the extent contains two blocks. Thus, after the execution, six blocks will be used, and the last one will be empty. Now, in the third step, remove the data of the third tuple. The third block will be associated with the table but will be totally empty. It is clear, that only four data blocks are relevant for the evaluation, just only they contain the same data portions. By index, they are accessible via *ROWID* values of the index. However, in this case, if the index does not contain the attributes characterizing the query condition, the *TAF* approach method is used. Unfortunately, the *TAF* method does not have any information about the empty blocks associated with the table, no data defragmentation or migration is

KVET, M., ŠALGOVÁ, V., KVET, M., MATIAŠKO, K. (2019). *Master Index Access as a Data Tuple and Block Locator*. Proceedings of the 25th Conference of Open Innovations Association (FRUCT).

done due to performance impacts – such a table would be inaccessible during such process, which is not acceptable. Moreover, nowadays, the number of update statements is high and is still rising, thus, data consolidation would require to be executed too often to ensure the benefit, but it is too resource demanding. Overall, the improvement would be minimal, even if any. Thus, by using *TAF* in the described situation, six blocks would be loaded into the memory, but two of them do not provide any data. The global efficiency would be 4/6 – just a bit higher than 66%. Sure, it is just a simple demonstration of the problem, in the real environment, performance significantly below 50% would be reached, so more than half of the system work would be unnecessary for the evaluation and the processing. This is, of course, a huge problem in terms of the performance and growth of the data requirements and complexity. Individual processing steps are shown in Figure 8.1.

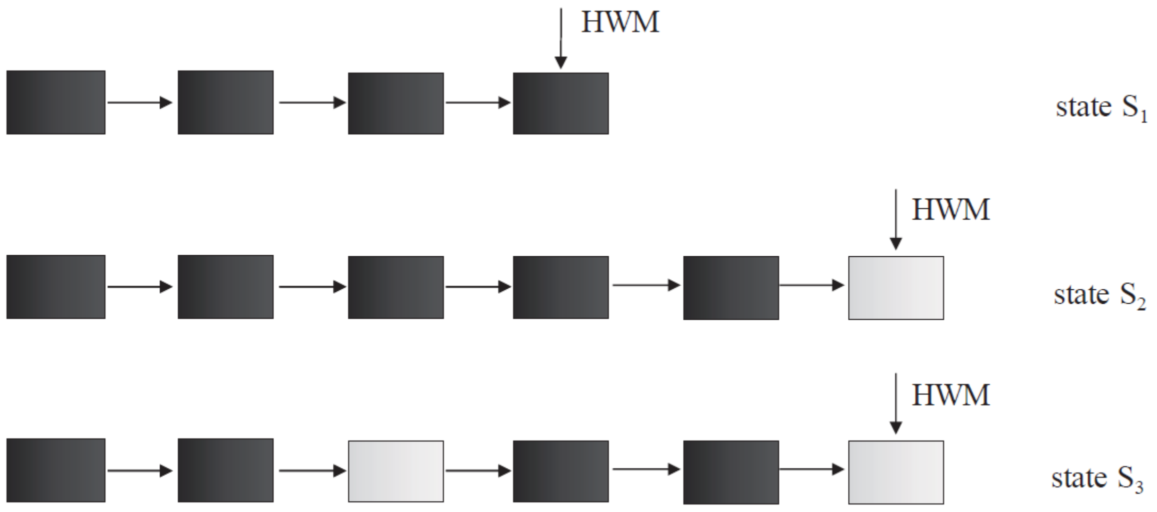


Fig. 8.1: Data block modeling

Our solution is based on the *Master Index*. It uses the fact, that each data row in the relational database can be uniquely identified using the primary key, thus each table usually contains a primary key definition, which has the property of the *uniqueness* and the *minimalism*. Whereas the value of the primary key must be present from the definition (cannot hold undefined NULL value) and the primary key automatically creates the index, in the system, at least one index exists with ROWID pointers to each data are present inside. From this point of view, if the index would be used, just the relevant blocks would be selected. The solution is shown in the data flow diagram in Figure 8.2. When the query is obtained to be evaluated, first of all, existing index suitability is evaluated. If there is a suitable index, naturally, it is used. In this case, therefore, there is no change compared to existing approaches. One of the *Index scan* methods is used, either *Index unique scan*, *Index range scan*, or *Index full scan* with its variants. If there is, however, no suitable index based on the data characteristics and conditions, our solution as the data optimizer extension is used. The system evaluates whether there is a Master index definition for the particular table. If not, the *Table Access Full* method must be used with all its limitations. However, if one of the indexes is so marked, it will be used, not for the data evaluation, just as data access [41].

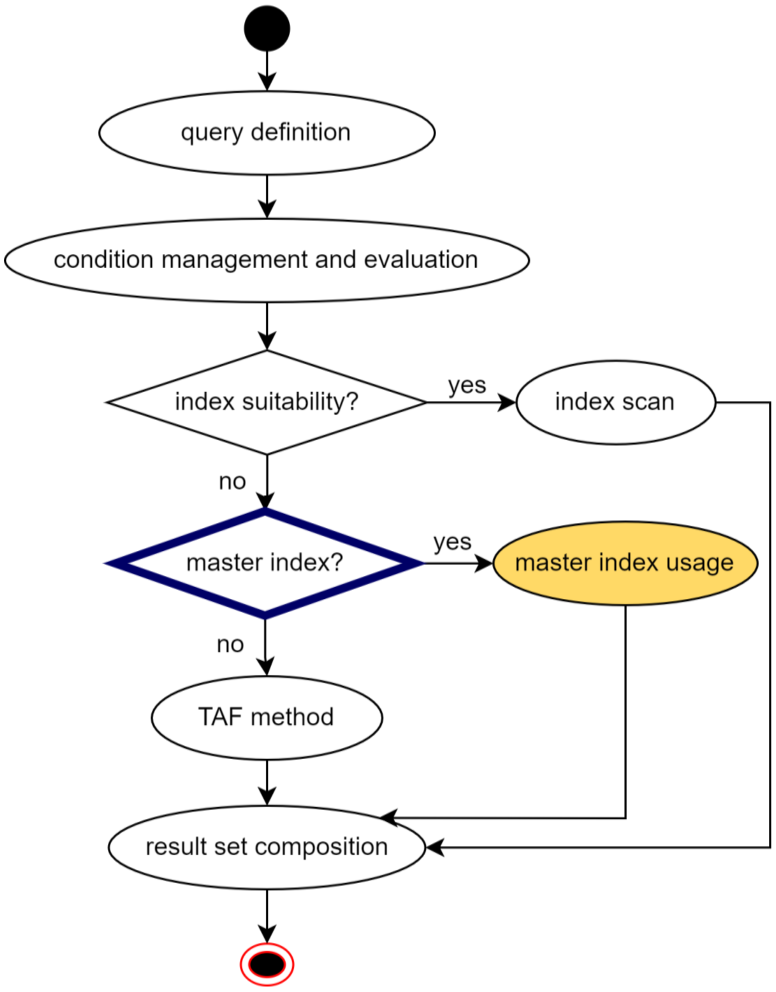


Fig. 8.2: Data flow - Index access selection

8.1.1 MASTER INDEX DEFINITION

There is only one strict requirement for the *Master index definition (MID)* – all data rows must be accessed via it. Thus, it must cover all the data. As mentioned, most often relational database index structure is *B-tree*, respectively *B+ tree*. It has one limitation – undefined (NULL) values are not indexed. So, if at least one of the indexed columns has the property of potentially holding NULL value, there is no certitude, that all the data are present by using an index. However, whereas in principle, each table is delimited by the primary key definition, such a suitable index should always be present. The point is just, whether it is the best suitable or not.

The Master index definition is defined for each table and can be selected either automatically or manually based on the user´s decision. The decision for the table can be selected as follows:

```
ALTER TABLE <table_name> SET MID = <index_name>;
```

In the previous case, the user defines the *Master index* manually. Thus, if the index is dropped or denoted as corrupted (needs to be rebuilt), the *MID* parameter is automatically set to NULL and the proposed technology will not be used later. Thus, if the setting for the table would be NULL, the proposed extension would not be applicable for the table resulting in using the original *TAF* method.

```
ALTER TABLE <table_name> SET MID = NULL;
```

Selection of the *Master index* can be done automatically by the system, as well. The decision is done by the optimizer based on the current statistics of the index and the whole system. The suitability of the index is to be declared by its size on the leaf layer. Generally, a fewer number of nodes indicates better performance. Another aspect of the selection is just the availability of the index in the *Buffer cache* memory structure. Similar to the table, the index must be loaded into the memory to be processed, as well. If some index is already available there, although partially, the process of the loading using I/O operations is removed, respectively shortened. Therefore, it is gainful to use automatic system management and decision-making. The option is done on the table granularity by using the following command:

```
ALTER TABLE <table_name> SET MID = AUTO;
```

The advantage of this approach is reflected by efficiency. If some index is dropped, the system automatically evaluates, whether it is marked as *master* or not. If so, a new suitable index for the processing is selected, if possible [41].

8.1.2 INDEX MASTER METHOD

In the previous paragraph, the principle of *Master index* definition selection is described. For now, it is necessary to explain the principle of data access. *TAF* method principle is characterized by the sequential scanning of all data blocks associated with the table. It uses the fact, that the individual blocks are formed in the extent shape, which is linked together. As described, for the processing and evaluation, the block is always loaded into the memory, even if it does not include relevant data for the query, respectively, it is empty. Thus, in the first phase of the development of the own approach, the aim was to remove such blocks from the evaluation. Our firstly defined solution was based on two sides linked list. Each block then consisted of the information about the fullness of the direct following block (***model 1***). Thank to that the empty block is skipped from the evaluation. The disadvantage is the necessity to store the two-way linked list and modification of the whole path after the change on the block level, as well. Our second proposed solution (***model 2***) improved the original approach by storing the pointer to the next used data block. The principle is shown in Figure 8.3. Black block is occupied, white is free. Let´s assume, that the third and last associated blocks are empty. Bold arrows indicate added pointers to the other blocks. Whereas the last block is free, the fifth can point either to the same (fifth block) or NULL pointer can be used regarding the consecutive way of evaluation. NULL is better from the size point of view but worse for the management.

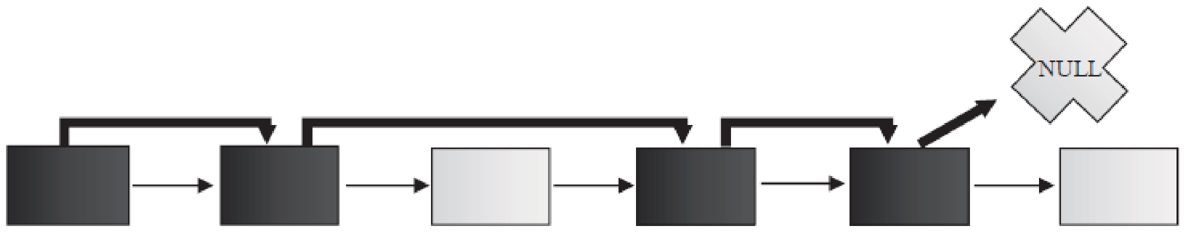


Fig. 8.3: Principle of storing the pointer

After the complex experiments, we concluded, that the proposed solution is robust, but not optimal. Inside each block, specific space had to be allocated for the pointers and fullness management. As a consequence, each block itself had to be shortened based on the size. Therefore, we also define another solution (***model 3***), which is just based on the *Master index*. It is used as the source of pointers to the data blocks. Is it possible to determine existing and at least partially occupied blocks? Sure, it is, by using ROWID pointers at the leaf layer. To ensure efficiency, we added a new parameter associated with the *MID*. It holds the pointer to the first node at the leaf layer of the index. Thanks to that, it is not necessary to traverse the whole index from the root. The name of the parameter is *MID_pointer_locator* and is maintained automatically, thus if the structure of the index is changed as the result of rebalancing, such parameter is automatically notified to ensure correctness.

MID_pointer_locator gets the first index node for the processing, respectively the first ROWID pointing the block inside the database. B+ tree has a linked list on the leaf layer, therefore individual data segments can be directly located from that level. Logically, there is a list of ROWIDs, which are used to access the physical data. Thus, non-relevant data blocks are not processed at all, whereas no ROWID points to them.

Our proposed approach uses the *Private Global Area (PGA)* of the server associated with each session separately. In this structure, local variables are stored. In our case, we use it for the list of individual blocks. Multiple rows can be located in the same data block. Therefore, before the evaluation, the address of the block is extracted from the ROWID value, which is consecutively checked, whether such block has already been processed or not. The whole block is evaluated, not only the row itself. The reason is based on the efficiency of I/O operations. It could happen that a block with multiple records is read into the memory. If only one record was evaluated, such a block would have to be processed later for further records. In the meantime, however, the block could be removed from the *Buffer cache* as it is a *clean* block type – no changes were made to it. Thus, the number of I/O operations would increase beyond the number of blocks actually used. The diagram expression of the processing steps is shown in Figure 8.4.

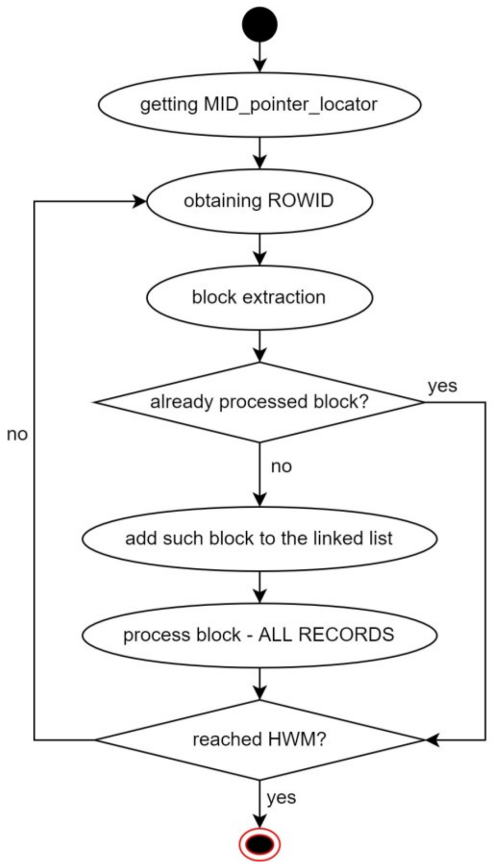

getting MID_pointer_locator
obtaining ROWID
block extraction
already processed block?
yes
no
no
add such block to the linked list
process block - ALL RECORDS
reached HWM?
yes

Fig. 8.4: Data flow – MID_pointer_locator and consecutive data management

8.1.3 ROWID VS. BLOCKID

In the previous definition, the principle of using ROWID values to identify blocks was used. As described, management was extended to check, whether such block had already been processed or not. The aim of the solution was clear, to minimize the number of I/O operations, which is part of the most expensive operations of the systems themselves. As a result, it would be grateful, if the solution can use identifiers of the block on the leaf layer instead of the ROWIDs. It is, however, not possible directly, whereas there is no possibility to modify existing index approaches in the core of the database system. The solution is, therefore, based on two interconnected index structures. One of them resides in the original and consists of the ROWID values in the leaf layer. The difference is that they do not point to the data blocks in the physical database but are routed to the second index. Pointers are always paired – from the index to the block module and vice versa as well. Thanks to that, any change in the block management can be easily identified and the whole supervising layer can be modified. Block module form is similar to the index; it uses the B+ tree structure too. On the leaf layer, pointers to the physical database are on the block granularity. If any block is freed, respectively associated without particular data, such blocks are not part of the *block module* and are automatically skipped. *The Master index* method uses only the block module and scans the blocks in a parallel manner. If there is any change in the data, the original index is used, which, however, automatically reflects the change in the block module, if any change in the segment or extend block positions are done. *Select* statements use direct access to the block module. The architecture of the solution (***model 4***) is in Figure 8.5.

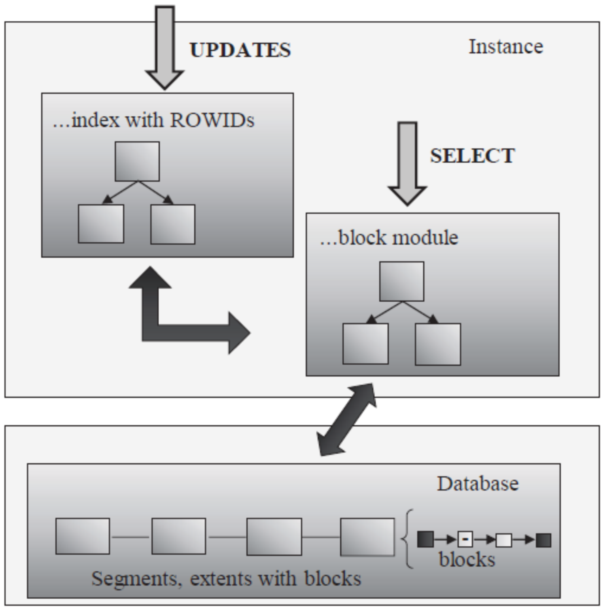


Fig. 8.5: The architecture of the solution

8.1.4 RESULTS

Performance characteristics have been obtained by using Oracle 18c database system based on the relational platform. For the evaluation, a table containing 10 attributes was used, delimited by the composite primary key consisting of two attributes. No specific indexes were developed, therefore the primary key was denoted as the Master index.

Experiment results were provided using Oracle Database 18c Express Edition Release 18.0.0.0.0 – Production Version 18.4.0.0.0 – Production. Parameters of the used computer are:

- Processor: Intel Xeon E5620; 2.4GHz (8 cores),
- Operation memory: 16GB (8 modules, DDR 1333MHz),
- HDD: 500GB.

Results and performance solution management were applied to the models described in the previous paragraphs. Four models were evaluated. *Model 1* extends the block definition by the information about the fullness of the direct following block. The negative aspect was identified, if multiple blocks in the linked chain were free, located together in the group. *Model 2* removes such constraint and contains the pointer to the following used block with relevant data. As evident from the results, the proposed solution brought relevant improvement in terms of processing time and size of the whole structure, as well.

Model 3 uses different architecture. It does not modify the physical structures inside the database block but uses our proposed *Master index* approach. Thus, if the index definition is not suitable, the marked Master index is used to locate the data. This approach is very convenient if the data tuples are modified very often with various size demands for the attribute values. Solution plays a significant role also in cases of data fragmentation in the database structure. Improvement of the solution is carried by the last *model 4*.

Obtained results are shown in Figure 8.6, which reflects the performance expressed by the processing time in the second precision. Values in the graph express the improvement or showdowns in the processing time in percentage. The referential model uses the original method for data access – *Table Access Full*. As evident, all of them offer significant improvement. 10% of the data were part of the result set. In optimal conditions, a perfectly defined index for the query would require 10% of the processing time in comparison with the *TAF* method. In our case, *model 1* obtained 23% load, *model 2* required 21%. Significant improvement was reached when using *model 3* – 17%. Architectural *model 4* was the best and required only 13%. Thus, 3% of the processing was associated with *Master index* management with an emphasis on the data location in the leaf layer. The optimal solution would require no data fragmentation, which is, unfortunately, very difficult to ensure in the real system environment, where the structure and size of attribute values can vary significantly. As a consequence, real deployment would degrade to use fixed-size variables, mostly strings, which, of course, is not entirely appropriate for disk space requirements.

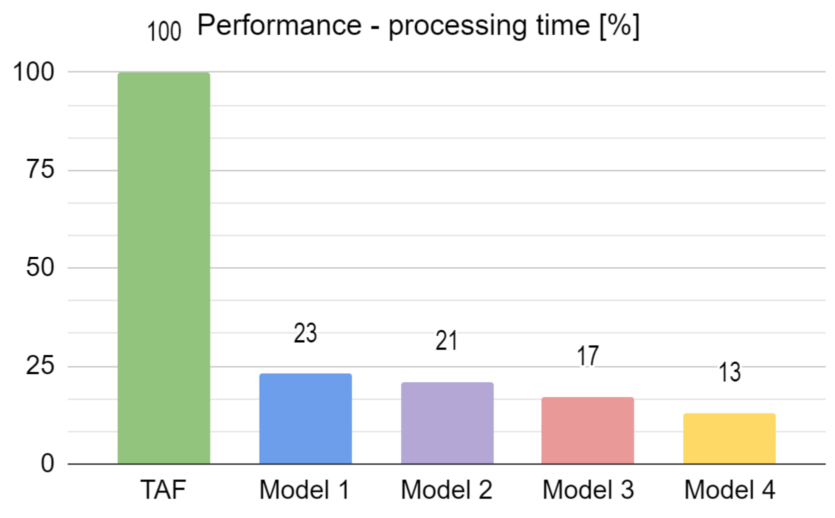


Fig. 8.6: Processing time results

When dealing with the size demands for the whole structure, the following results were obtained. The original solution with no specific structure and management added was used as referential. Values are in percentage expressing the additional demands. Results are in Figure 8.7. *Model 1* reduces the size of the block itself to extend the header to store information about the next block and load it. It required additional demands of 5%. *Model 2* uses only pointers, which do not need to indicate a direct following block if it is empty. It removes the impact of the free block grouping, as well. It required just 3% of additional size demands. Slight differences using only 0,1% can be identified in *model 3*. It does not, namely, use any additional structure, just one of the indexes meeting the requirements is marked as *Master*. *Model 4* is the most complicated, whereas an additional index on block granularity is used. In this case, the size requires an additional 12%.

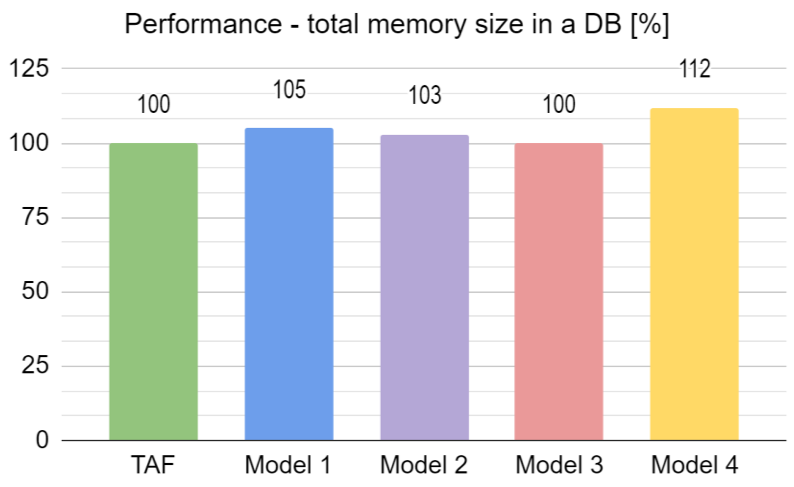


Fig. 8.7: Size demands

The last experiment in this section is based on evaluating the rate between block occupation and free blocks caused by data restructuralization, shifting, and fragmentation. Results are shown in Figure 8.8 expressing the rate of actually used blocks (block, which consists of at least one relevant tuple inside). Again, we use our four designed models, that are paired to the original *TAF* access method. Based on the results, the suitability of *model 1* is limited by the value of 63, *model 2* limitation is 65. *Model 3* can work effectively up to 81 and *model 4* – 84. These values express the rate between free and used blocks. Thus, the best performance was obtained by *model 4*, which can work effectively up to 16% (100 - 84) of free blocks. If the rate is less (the number of free blocks is lower than 16%), the original *TAF* is more effective, although nonrelevant blocks must be scanned. If there is no free block in the system – all of them have at least one consistent data tuple, reached results are following (expressing the slowdown of the system in terms of the processing time). Values are expressed in percentage:

• Model 1	58%,
• Model 2	51%,
• Model 3	23%,
• Model 4	23%.

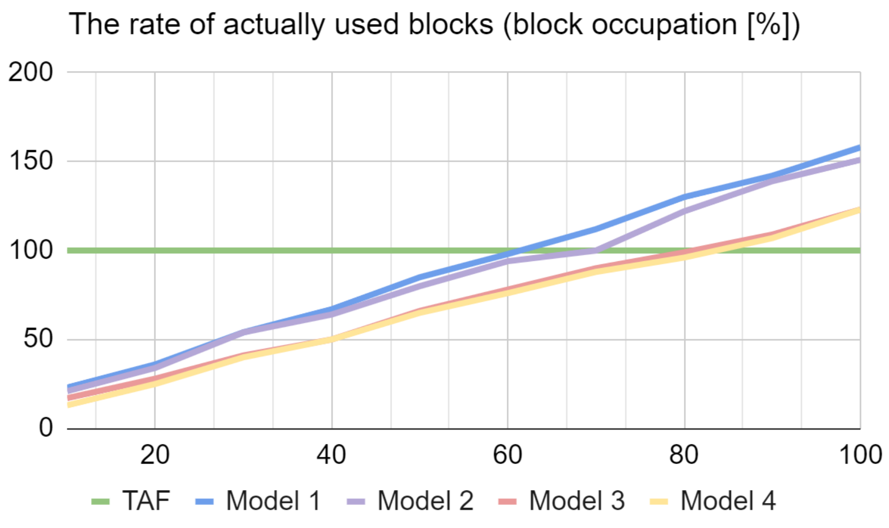


Fig. 8.8: Block rate results

The proposed solution has been tested in the DBS Oracle environment but can be adapted to any relational system, whereas the principles are the same. We assume that DBS Oracle is the most comprehensive and powerful system and technology at the same time.

8.1.5 SUMMARY

Effectivity of data processing is one of the most significant tasks to ensure the performance of the whole system. Nowadays, the number of data to be handled is high and is still rising. The structure of the data can evolve, as well. Moreover, such data dynamically change their values and properties over time. As a consequence, table complexity is still rising, and more and more new data fragments are identified. Therefore, it is clear, that these factors must be taken care of when the database system management is defined. Many systems can be connected to the same database producing various data analysis. Thus, data queries can vary significantly resulting in poor performance, whereas it is not possible to develop all indexes for the defined properties. Sequential scanning of all blocks associated with the table is the last step before the total collapse of the system and user as well, whereas they are not willing to wait for the result sets. The aim of our proposed solution is to limit the necessity to use sequential data block scanning performed by the *Table Access Full (TAF)* method. Our technology uses the *Master index*, which is not used for the evaluation itself, whereas it does not fit the conditions of the query. The core is that it contains all the pointers to the data on the leaf layer. Therefore, the index itself is used as the data locator. There are two proposed models highlighting the *Master index* definition. The first one is based on data row granularity, and the second one is shifted to block identification and uses two indexes. Based on the reached results, the best solution reflects block granularity. In this case, in comparison with the original *TAF* method, performance in processing time was lowered to 13%, if one-tenth of the data should be provided in the result set. The principle is based on removing the evaluation of free blocks, which must be transferred into the memory in a standard manner. Vice versa, there is an approximately 12% increase in size demands. It is caused by the necessity to develop a new index on the block granularity for the queries.

8.2 REDUCING DATA ACCESS TIME USING PARTITIONING

Various partitioning techniques and methods were described in chapter 5. We have used them to perform experiments aimed at comparing data access time based on different types and numbers of created partitions.

8.2.1 RESULTS

Data access time characteristics have been obtained by using the Oracle 18c database system based on the relational platform.

Experiment results were provided using Oracle Database 18c Express Edition Release 18.0.0.0.0 – Production Version 18.4.0.0.0. Parameters of the used computer are:

- Processor: Intel(R) Core(TM) i5-3317U; 1.70GHz,
- Operation memory: 8GB,
- HDD: 500GB.

8.2.2 RANGE PARTITIONING

To compare the data access time, two tables were created. Both tables had identical data and contained 1,000,000 rows with 3 columns of Integer, Date, and Varchar2(20) data types. One table was created without partitions and the other one was divided into 34 partitions according to *the range partitioning technique*, where each partition represented one

ŠALGOVÁ, V., MATIAŠKO, K. (2020). *Reducing Data Access Time using Table Partitioning Techniques*. 18th International Conference of Emerging eLearning Technologies and Applications (ICETA).

month of the specific year. Data were generated into partitions evenly, so each partition had approximately the same number of rows. The experiment consisted in selecting data from such date intervals that the data were selected only from a certain number of partitions. We started consecutively accessing one part of the table, then added other ones, until we finished accessing data from all parts of the table. Each of the steps was executed 20 times to avoid inaccuracies and to calculate the average value. Access times of these two tables can be seen in Table 8.1.

PARTS ACCESSED	TABLE WITH 34 PARTITIONS	TABLE WITHOUT PARTITIONS
1	0,0261	0,0813
2	0,0236	0,0075
4	0,0215	0,0641
8	0,0196	0,0596
14	0,0217	0,0581
16	0,0193	0,0443
20	0,0223	0,0369
25	0,0208	0,0284
26	0,0205	0,0280
27	0,0214	0,0266
28	0,0206	0,0238
29	0,0213	0,0224
30	0,0206	0,0202
31	0,0225	0,0201
32	0,0211	0,0204
33	0,0209	0,0198
34	0,0246	0,0208

Tab. 8.1: Access time - Range partitioning

From the data of Table 8.1, the graph in Figure 8.9 was constructed, according to which it is obvious that the table with the 34 partitions created had a significantly shorter data access time from the beginning. It was only 0.0261 seconds, which represents 32.1% of 0.0813 seconds of the access time of the table without partitions. According to the results, it is clear that the creation of partitions is suitable if only a smaller group of data is approached compared to access to all data in the table. Namely, when selecting data from a larger interval, which represented a larger number of partitions, it gradually became more advantageous to access the table without partitions, because the data could be accessed sequentially and each partition did not have to be accessed sequentially, which is ultimately slower than sequential access. According to our experiment, we can see that the data access time in the table without partitions decreased so much when selecting data from a gradually larger interval that when accessing data representing 30 partitions out of 34. This time was even lower than when dividing data into partitions. In our experiment, these 30 out of 34 partitions represent the limit when the creation of partitions is no longer optimal in terms of data access time. From this point, access to data without partitions has been confirmed as faster in the remaining small number of cases.

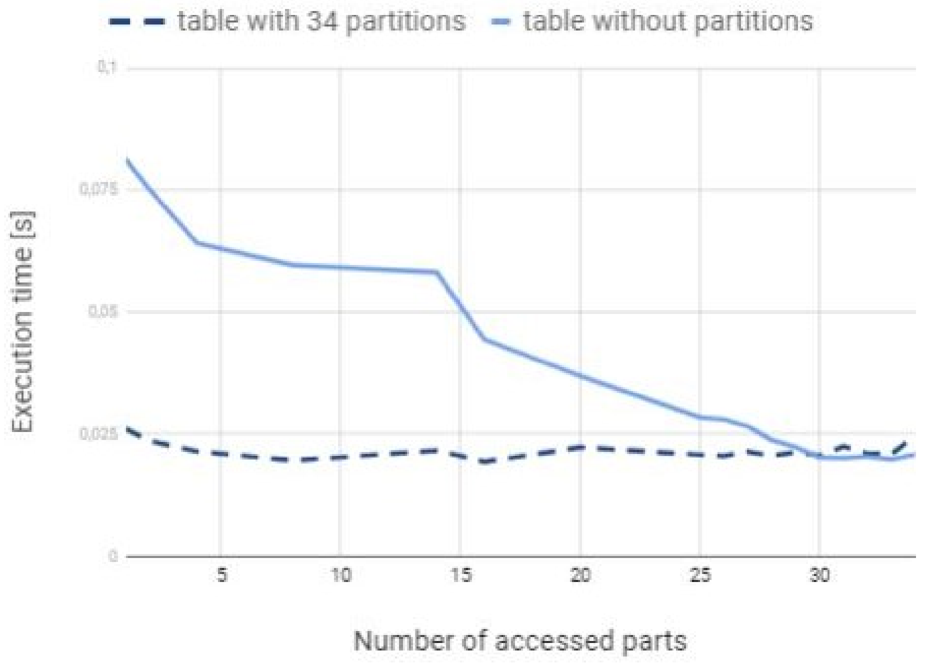


Fig. 8.9. Access Time - Range Partitioning

8.2.3 LIST PARTITIONING

In an experiment aimed at comparing data access times of a table without partitions and a table with partitions according to list partitioning, two tables with identical data were created. Each table contained 1 000 000 rows with 3 columns of Integer, Varchar2(20), and Varchar2(10) data types. One table did not contain any partitions and the other was divided into 12 partitions according to the listed values of the state names, where each partition was described by three values. Thus, the list partitioning technique was used. Data were generated into partitions evenly, so each partition had approximately the same number of rows. The experiment consisted in selecting data from such date intervals that the data were selected only from a certain number of partitions. We started consecutively accessing one part of the table, then added other ones, until we finished accessing data from all parts of the table. Each of the steps was executed 20 times to avoid inaccuracies and to calculate the average value. Average access times can be seen in Table 8.2.

PARTS ACCESSED	**TABLE WITH 12 PARTITIONS**	**TABLE WITHOUT PARTITIONS**
1	0,0170	0,1942
2	0,0200	0,2139
3	0,0220	0,2190
4	0,0261	0,2290
5	0,0274	0,2283
6	0,0312	0,2411
7	0,0316	0,2407
8	0,0391	0,2419
9	0,0341	0,2493
10	0,0380	0,2534
11	0,0373	0,2655
12	0,0368	0,2810

Tab. 8.2: Access time - List partitioning

From the data in Table 8.2, the graph in Figure 8.10 was created, which shows a significantly improved access time to data from the table with 12 partitions using *the list partitioning technique*. Unlike the range partitioning experiment where there were a few exceptions when accessing a lot of table parts, the list technique showed significantly better access time for all options of the number of parts accessed. It means, even when selecting data from all the table partitions. When accessing all partitions, the access time was only 0.017 seconds, which is less than 9 percent of the access time to the data in the table without partitions.

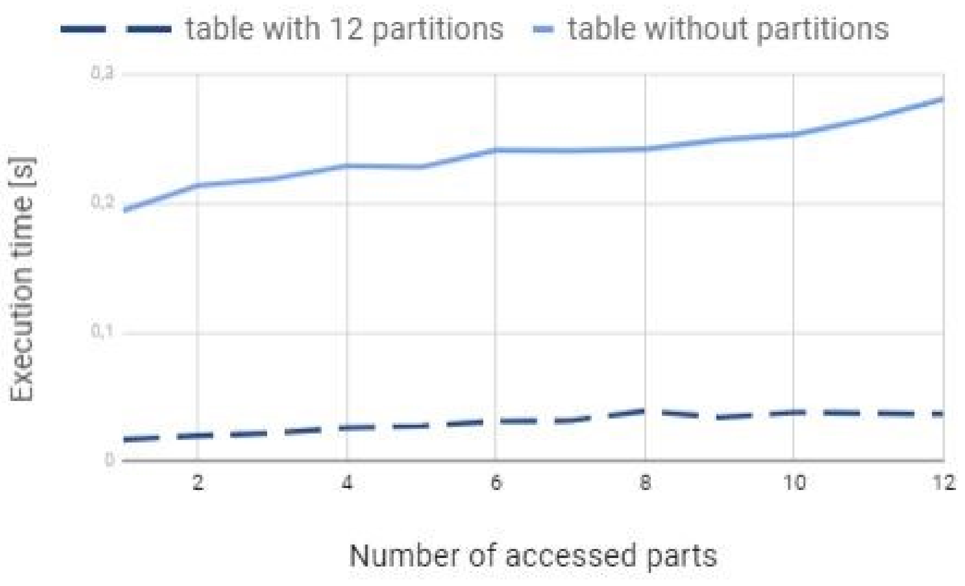


Fig. 8.10: Access Time - List Partitioning

8.2.4 SUMMARY

The efficiency of data access is one of the most important tasks in ensuring system performance. The amount of data is constantly growing, and therefore this data needs to be divided in some way. When using large amounts of data, great emphasis is placed on access time. For this reason, the creation of partitions and the subsequent division of data into them can bring a significant improvement in the time of access to the data.

In the experiments, the access times for the data in the tables with the created partitions and tables without any partitions were compared. The results significantly showed that access to data stored in partitions can significantly reduce the time of access to data and thus bring greater efficiency. The use of the range partitioning technique provided a significantly improved access time when less than 30 of the 34 partitions were accessed. With access to 30 or more partitions, data storage without partitions has proven to be more appropriate. When using list partitioning, there was an even more significant improvement in performance time, and this improvement persisted even when accessing multiple or all partitions. In conclusion, it can be evaluated from the results that partitioning can significantly improve data access time.

8.3 EFFECT OF PARTITIONING AND INDEXING

We have dealt with the effect of partitioning and indexing on data access time, which was compared in seven different scenarios of different combinations of partitions created over tables and indexes.

8.3.1 METHODOLOGY

Data access time characteristics have been obtained by using the Oracle 18c database system based on the relational platform.

Experiment results were provided using Oracle Database 18c Express Edition Release 18.0.0.0.0 – Production Version 18.4.0.0.0. Parameters of the used computer are:

- Processor: Intel(R) Core(TM) i5-3317U; 1.70GHz,
- Operation memory: 8GB,
- HDD: 500GB.

To compare the data access time, two tables were created. Both tables had identical data and contained 1,000,000 rows with 4 columns of Integer, Varchar2(30), Date, and Integer data types. One table was created without partitions and the other one was divided into 25 partitions according to the *range partitioning* technique which was applied to the last integer column storing data about percentages. It means that each partition contained a range of 4 distinct percentage values. Data were generated into partitions evenly, so each partition had approximately 40,000 rows. A non-partitioned table was created as follows:

```
CREATE TABLE table_not_partitioned (id INT PRIMARY KEY,
                                    name VARCHAR2(30),
                                    dateB DATE,
                                    percents INT);
```

ŠALGOVÁ, V., MATIAŠKO, K. (2021). *The Effect of Partitioning and Indexing on Data Access Time.* Proceedings of the 29th Conference of Open Innovations Association (FRUCT).

A table with 25 partitions was created as follows:

```
     CREATE TABLE table_partitioned (id INT PRIMARY KEY,
                                     name VARCHAR2(30),
                                     dateB DATE,
                                     percents INT)
       PARTITION BY RANGE (percents) (
         PARTITION p1 VALUES LESS THAN(5),
         PARTITION p1 VALUES LESS THAN(9),
          …
         PARTITION p24 VALUES LESS THAN(97),
         PARTITION p25 VALUES LESS THAN(101)
       );
```

Situations without created indexes were tested over each of these two tables. After that, different types of indexes were created for these tables as well. The different scenarios concerned a non-partitioned and a partitioned index.

Partitioned indexes were created as global, local prefixed, and local nonprefixed indexes. In the case of partitioned indexes, the partitions were created in the same way as in the tables, and thus they related to a percentage column that was divided into 25 ranges. Seven executed scenarios are shown in Table 8.3. Each scenario indicates whether the table was partitioned or not, and which type of index was created on the table.

	TABLE	**INDEX**
1	nonpartitioned	-
2	partitioned	-
3	nonpartitioned	partitioned (global)
4	nonpartitioned	nonpartitioned
5	partitioned	nonpartitioned
6	partitioned	partitioned (local prefixed)
7	partitioned	partitioned (local nonprefixed)

Tab. 8.3: Executed scenarios

The global partitioned index in scenario no. 3 was created on the attribute *percents* as follows:

```
CREATE INDEX index_global_percents
  ON table_not_partitioned(percents) GLOBAL
    PARTITION BY RANGE (percents) (
      PARTITION p1 VALUES LESS THAN(5),
      PARTITION p1 VALUES LESS THAN(9),
       …
       PARTITION p24 VALUES LESS THAN(97),
       PARTITION p25 VALUES LESS THAN(MAXVALUE)
    );
```

The non-partitioned index in scenarios no. 4 and no. 5 was created in a similar way for both of the tables on the attribute *percents* as follows:

```
CREATE INDEX index_percents_nonpartitioned
  ON table_not_partitioned(percents);
```

The local prefixed partitioned index in scenario no. 6 is partitioned on a left prefix of the index columns and was created on the attribute *percents* as follows:

```
CREATE INDEX index_local_pref_percents
  ON table_partitioned(percents) LOCAL;
```

The local non-prefixed partitioned index in scenario no. 7 is not partitioned on a left prefix of the index columns. It was created on the attribute *dateB* and *percents* as follows:

```
CREATE INDEX index_local_nonpref_percents
  ON table_partitioned(dateB, percents) LOCAL;
```

8.3.2 RESULTS

The data access time of executed seven scenarios is shown in Table 8.4 in milliseconds. The left column represents the number of accessed ranges.

	S(1)	S(2)	S(3)	S(4)	S(5)	S(6)	S(7)
1	52	4	5	6	6	6	6
2	53	6	7	8	9	12	9
3	51	7	9	13	16	15	18
4	54	9	14	16	16	18	20
5	58	14	19	25	28	26	25
6	60	16	23	33	32	29	29
7	59	20	26	39	60	29	33
8	61	23	35	61	62	27	36
9	60	24	33	62	64	26	38
10	58	27	41	66	67	30	39
11	61	32	38	63	69	31	42
12	61	67	40	66	65	32	45
13	60	566	43	68	64	64	49
14	59	619	45	70	67	39	50
15	61	672	50	75	65	41	52
16	55	732	53	66	68	59	55
17	58	797	54	66	66	54	60
18	60	805	61	69	69	51	61
19	59	920	63	70	70	49	65
20	58	861	59	71	67	54	74
21	56	981	63	73	68	51	70
22	58	986	65	71	69	62	67
23	57	1049	64	72	71	1064	71
24	57	1070	69	68	72	1065	71
25	58	1153	69	67	72	1067	72

Tab. 8.4: Data access times of 7 scenarios

When comparing scenarios no. 1 and no. 2 in which there are tables without any indexes created, it can be concluded that when accessing data related to successively from 1 to 10 partitions, the access time was significantly shorter for a partitioned table. When selecting data for one partition, it was only 4 milliseconds compared to 52 milliseconds. The data access time of the partitioned table increased with the increasing number of accessed partitions. There was a change in access to 12 partitions. The partitioned table data access time was 67 milliseconds, which was more than the non-partitioned table data access time of 61 milliseconds. Thus, in this situation, the number of 12 partitions represented the limit when the partitioned table was no longer more advantageous than the non-partitioned table in terms of data access time. A comparison of scenarios no. 1 and no. 2 is shown in Figure 8.11 and Figure 8.12.

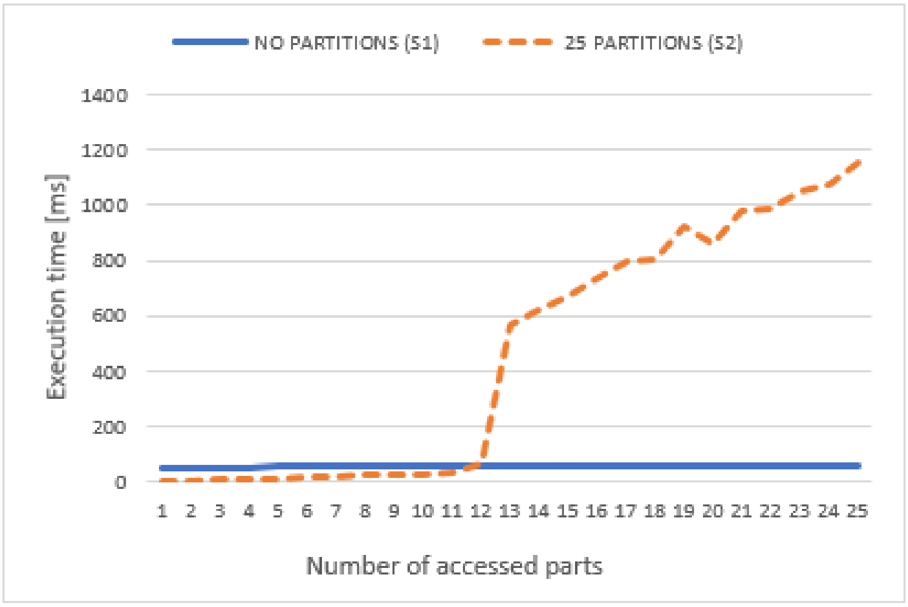


Fig. 8.11: Data access time – Scenarios 1, and 2

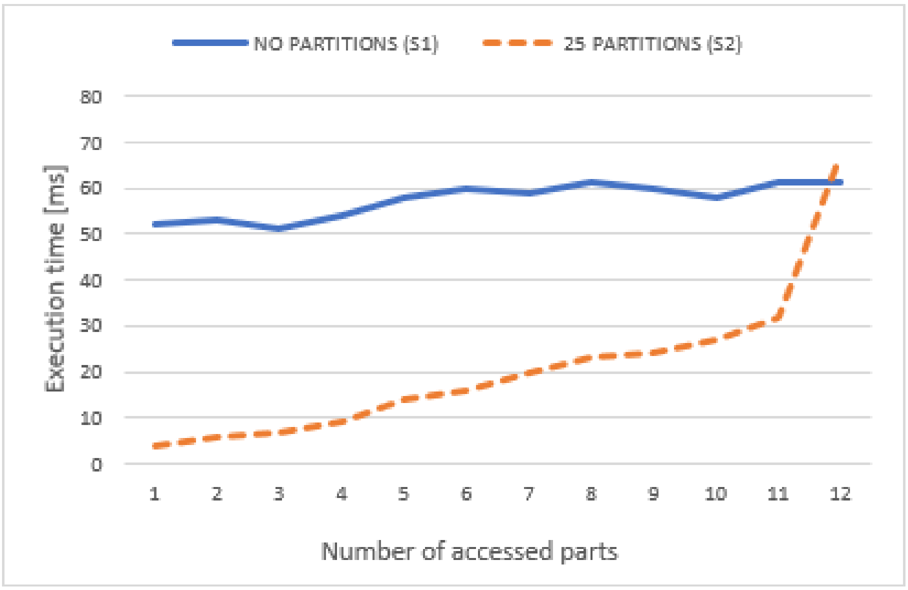


Fig. 8.12: Data access time – Scenarios 1, and 2 – Enlargement

Figure 8.13 shows data access times of scenarios no. 1, no. 3, and no. 4 concerning the table without partitions. Scenario no. 1, which was without any indexes, kept data access time between 51 and 61 milliseconds, without major fluctuations. Scenario no. 4, which concerned the nonpartitioned index, had a much shorter time from the beginning. When accessing data that would belong to one partition, the access time was only 6 milliseconds. With the increasing number of partitions that would be accessed, the access time gradually increased to 8, 13, 16, 25, 33, and 59 milliseconds, and the access time to the data in the range of 7 partitions was the same as in the scenario no. 1, for 61 milliseconds. Subsequently, the access time increased slightly from this limit. Scenario no. 3 started with an access time of 5 milliseconds and had the lowest time in most cases, up to access data that would cover 18 partitions. After this limit, the most advantageous scenario was the one without any indexes.

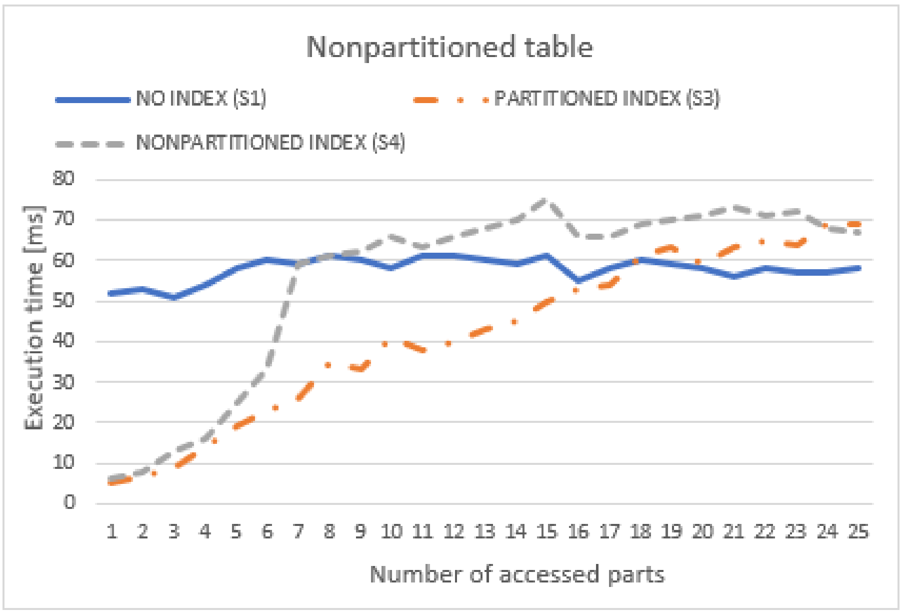


Fig. 8.13: Data access time – Scenarios 1, 3, and 4

The situations regarding the partitioned table are shown in Figure 8.14. Scenario no. 2, which was without any indexes, had the lowest time in the first half of the cases, compared to scenarios no. 5, no. 6, and no.7. Since the access to 11 partitions, the access time has grown significantly, and since the access to 12 partitions, scenario no. 2 had the largest data access time. Scenario no. 5, concerning the nonpartitioned index, had a much smaller increase. Its access times ranged from 6 to 72 milliseconds. The only major growth in time was between access to 6 and 7 partitions, where the time increased by 32 milliseconds to 60 milliseconds. Subsequently, the times did not have such large growth, they always increased by a maximum of 3 milliseconds. Scenario no. 6, concerning the local prefixed partitioned index, had access times to 1 to 22 partitions ranging from 6 to 62 milliseconds. Since access to the 23 partitions, the time has increased very significantly, reaching a time similar to that without any index, of more than 1060 milliseconds. Scenario no. 7, concerning the local nonprefixed partitioned index, had very similar access times as the nonpartitioned index scenario, in the same range of 6 to 72 milliseconds.

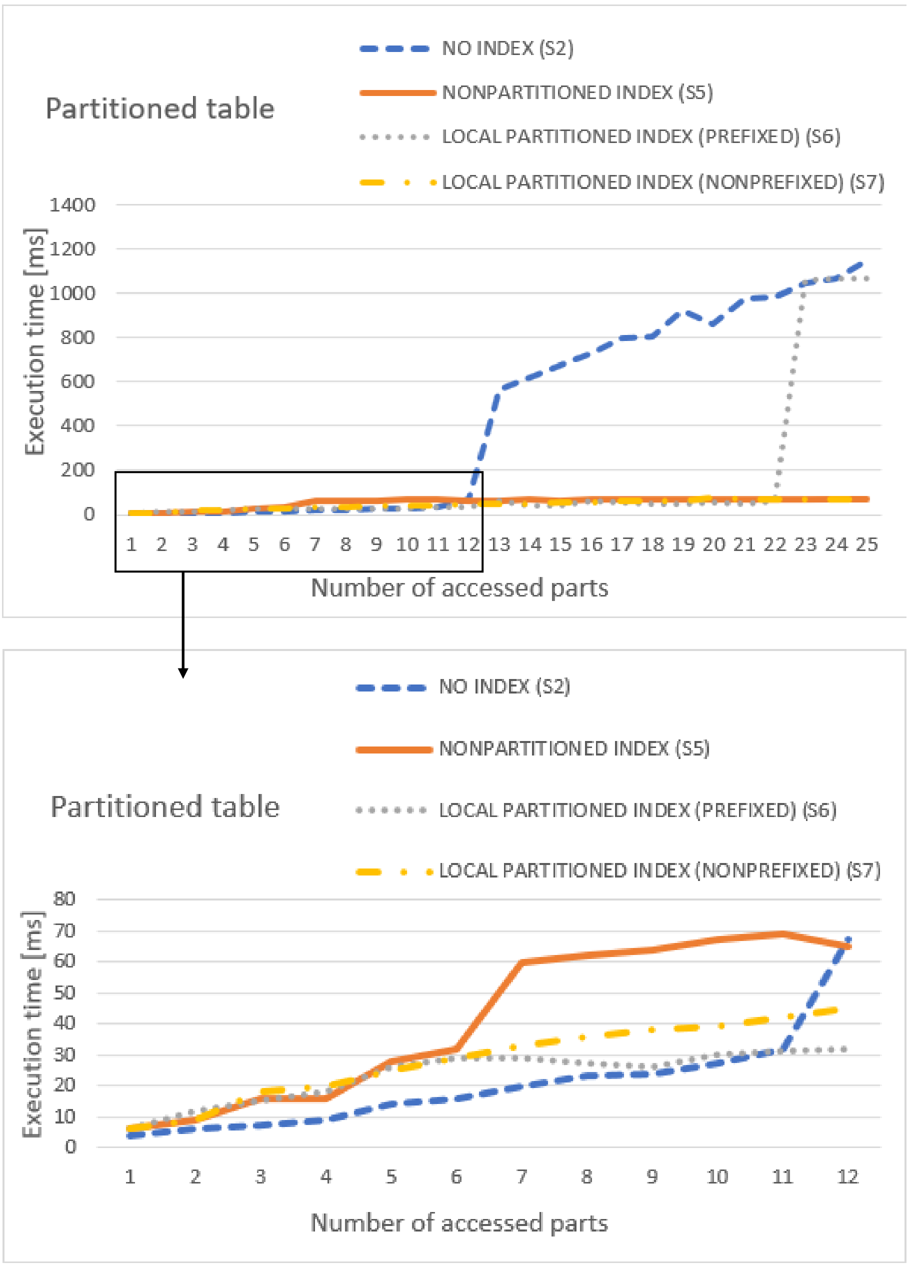


Fig. 8.14: Data access time – Scenarios 2, 5, 6, and 7

We also performed this performance evaluation study in the MySQL database system. Scenario no. 3, concerning a nonpartitioned table and a partitioned index, and scenario no. 5, concerning a partitioned table and a nonpartitioned index, were not performed because such creation in MySQL is not possible. Partitioning must be applied to all the data and indexes in a table, thus we cannot partition only the data and not the indexes, or vice versa. What is more, in MySQL there are no global partitioned indexes, all of them are made local. The performance evaluation study in the MySQL database system was performed on localhost with different computer parameters than in the Oracle experiment. Although it was executed and experimented on another machine, we can evaluate, that the individual improvement rates of executed scenarios remained analogous.

8.3.3 SUMMARY

The efficiency of data access is one of the most important tasks in ensuring system performance. The amount of data is constantly growing, and therefore it needs to be processed in an efficient way. When using large amounts of data, great emphasis is placed on data access time. Creating indexes, partitions and their various combinations can bring significant improvement in data access time in many situations.

In the experiments, the access times for the data in the table with the created partitions and the table without any partitions were compared in the DBS Oracle environment. We tested several situations that differed in the type of indexes created on both tables, such as non-partitioned, global partitioned, local partitioned prefixed, and local partitioned non-prefixed indexes.

The results showed that data access time differed a lot in various scenarios. For the first ranges of selected data, the worst access time was caused by using the non-partitioned table with no indexes. Here, the access time was at the beginning about 9-13 times worse than in the remaining scenarios. However, from about half of the accessed ranges had a significant slowdown in access time scenario with the partitioned table with no indexes and it started to be about 10 times slower than a scenario with the non-partitioned table with no indexes.

All scenarios in which partitioning, indexing, or their combinations were used started at approximately the same data access time of 4, 5, and 6 milliseconds. Regarding access to data belonging to the first 11 partitions out of 25, the scenario with the partitioned table with no indexes had the best results. However, after this limit of 11 partitions, it significantly became the slower scenario.

From the results of experiments, it can be deduced that the use of partitioning, indexes, or their combinations can significantly speed up data access time. However, with frequent access to a large amount of data to which a large number of partitions or index nodes are bound, the access time is similar to in the scenario without partitions and indexes, or in some situations, it may be even much worse.

Therefore, it is important to consider the appropriate choice of partition and index types, depending on the frequency and the volume of accessed data.

8.4 EFFECT OF INDEXES ON DML OPERATIONS

We have dealt with the effect of the number of indexes on the performance of the DML operations, such as insert, update, and delete.

8.4.1 DATA

The experiments were performed on a bike-sharing system. The part of the system concerning the bicycles rented by the customers was used, on which the individual operations were executed. The data model is shown in Figure 8.15.

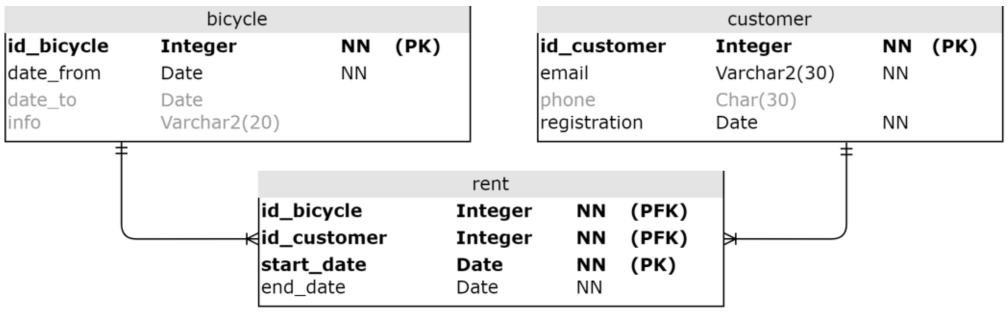


Fig. 8.15. Data model

Each of the entities for bicycles and customers contained 10 000 rows. The M:N relationship between them created an entity of rents containing 10 million rows.

8.4.2 RESULTS

Data processing time characteristics have been obtained by using Oracle 18c database system based on the relational platform. Experiment results were provided using the Oracle Database 18c Express Edition Release 18.0.0.0.0 – Production Version 18.4.0.0.0. Parameters of the used computer are:

- Processor: Intel(R) Core(TM) i5-3317U; 1.70GHz
- Operation memory: 8GB
- HDD: 500GB

ŠALGOVÁ, V. (2021). *Effect of indexes on DML operations.* 14th International Scientific Conference on Sustainable, Modern and Safe Transport (TRANSCOM).

In each situation of inserting, updating, and deleting data we executed a series of these operations. About one million rows were affected in the loop at each step. Six situations were tested for each operation. The difference between the situations was in the number of created indexes related to the modified data. We started with the situation without creating indexes and sequentially created one to five indexes. They were related to customer identifiers, bicycle identifiers, years of individual bicycle rents, or a combination of these attributes and were created as follows:

```
CREATE INDEX index1 ON borrowed_bicycle(id_customer);
CREATE INDEX index2 ON borrowed_bicycle(id_bicycle);
CREATE INDEX index3 ON borrowed_bicycle(id_customer, id_bicycle);
CREATE INDEX index4 ON borrowed_bicycle(to_char(start_date, 'YYYY'));
CREATE INDEX index5 ON borrowed_bicycle(id_customer,
                                         to_char(start_date, 'YYYY'));
```

Each situation of inserting, updating, and deleting data was performed ten times with the same data to avoid inaccuracies. The resulting times form the calculated averages for each step.

By adding a new record to the table, the new node is also added to the corresponding indexes. After that, the rebalancing of the tree structure index is performed. Based on the results in Figure 8.16, it is obvious that creating indexes related to the inserted data has a relatively significant effect on the insert execution time. While in a situation without indexes, inserting rows took 85.09 seconds, after creating one index, the time increased to 92.43 seconds, and for 5 indexes affected by inserted data, it took even up to 218.58 seconds.

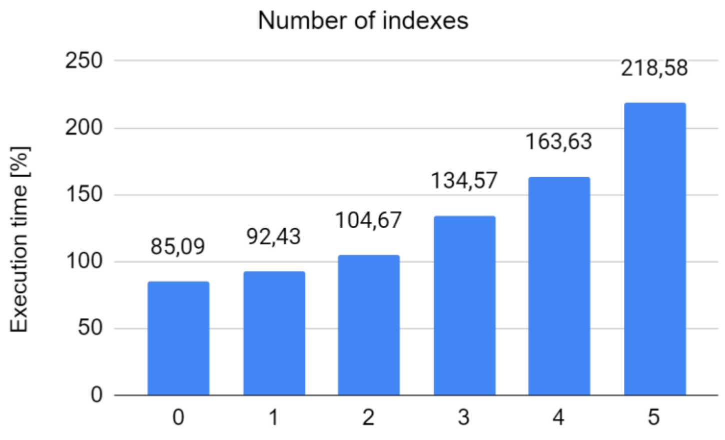


Fig. 8.16: The execution time of insert operations

Situations related to data updates are shown in Figure 8.17. Each of the update situations changed the start_date attribute of approximately 1 million rows of the borrowed_bicycle table. For example, for the borrowing with ID 1901, the update was executed as follows:

```
UPDATE borrowed_bicycle
  SET start_date = TO_DATE('13.01.2021', 'DD.MM.YYYY')
    WHERE id_borrowing = 1901;
```

The increase in the time needed to update the data in the case of no indexes compared to the case with the five indexes is significantly larger than when inserting data. Whereas in the situation without indexes the data update took 17.03 seconds, with 5 indexes it was up to 72.10 seconds. An update statement must relocate the corresponding index nodes to maintain the index order. For that, the old entry must be removed and the new one is added at the new location. The structure needs to stay balanced as well.

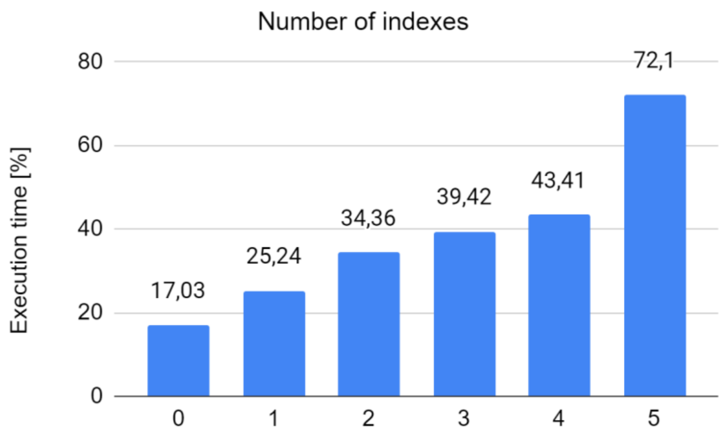


Fig. 8.17. The execution time of update operations

Figure 8.18 shows the results for data delete situations, which were executed as follows:

```
DELETE FROM borrowed_bicycle
  WHERE id_borrowing > 10000000;
```

The time required to perform delete operations increased to a similar extent as in the case of data updating. When deleting a row, the index deletes the row reference, and the tree index structure must be rebalanced in order to preserve the necessary property of the B-tree, respectively B+ tree.

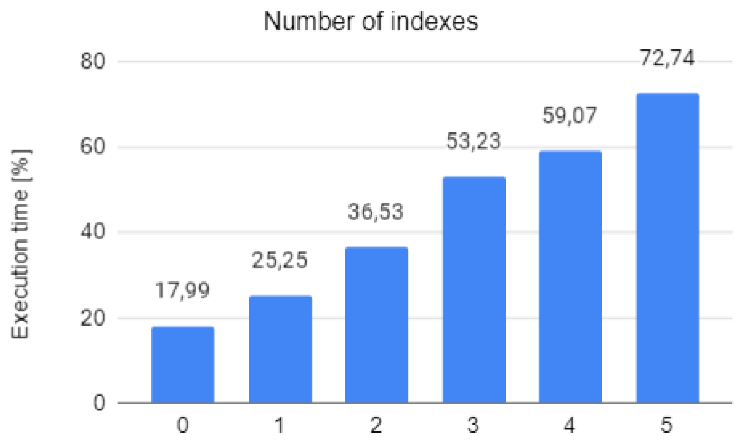


Fig. 8.18. The execution time of delete operations

8.4.3 SUMMARY

Indexes are very useful for speeding up data access time. However, from the resulting times of our experiments in Oracle, it can be concluded that the number of indexes makes indeed a significant impact on increasing the time to perform insert, update, and delete operations. It is therefore very important to establish by way of working with data an appropriate number of indexes. A larger number of indexes is only suitable for data that does not change often and is only used for retrieval. In situations with frequent and large data changes, it is better to use a smaller number of corresponding indexes.

8.5 THE IMPACT OF TABLE AND INDEX COMPRESSION

The amount of stored data is growing rapidly, and it brings considerable challenges. The enormous growth in the volume of data makes storage one of the biggest cost elements. For this purpose, relational databases are used very often. Fast access to data is becoming increasingly important and great emphasis is placed on its improvement. Many computer

---

ŠALGOVÁ, V. (2022). *The Impact of Table and Index Compression on Data Access Time and CPU Costs.* 10th World Conference on Information Systems and Technologies (WorldCIST).

systems heavily use compression nowadays. It is used for audio, image, and video data in multi-media systems, to carry out backups, compress inverted indexes in information retrieval, ship files across the Internet, and store large files and software packages. Data compression is widely used in data management to save storage space and network bandwidth and to reduce costs for storage media. Many query processing algorithms can manipulate compressed data just as well as decompressed data, and the processing of compressed data can even speed query processing by a factor much larger than the compression factor. Database performance strongly depends on the amount of available memory. It means all available memory should be used as effectively as possible and to keep and manipulate data in memory in a compressed form [12].

We have dealt with the impact of table and index compression on data access time and CPU costs. It was compared in eight different scenarios of different combinations of compressed and uncompressed indexes over compressed and uncompressed tables with non-unique data of various row numbers.

8.5.1 DATA COMPRESSION

Reducing the disk space requirements for a whole table or individual table partitions can be ensured by using compression. The innovative compression technique of Oracle RDBMS reduces the size of relational tables and is able to compress data much more effectively than standard compression techniques. Unlike other compression techniques, Oracle incurs virtually no performance penalty for SQL queries accessing compressed tables. The compression may provide performance gains for queries accessing large amounts of data, as well as for certain data management operations like backup and recovery.

The compression can reduce the amount of disk space required for a given data set. It has consequences on I/O performance. Firstly, the seek distances and the seek times are reduced because the reduced data space fits into a smaller physical disk area. Secondly, there can be more data on each disk page, track, and cylinder, allowing more intelligent clustering of related objects into physically near locations. Thirdly, there is an opportunity for disk shadowing at the unused disk space to increase reliability, availability, and I/O performance. What is more, compressed data can be transferred faster to and from the disk. Data compression is an effective means to increase disk bandwidth by increasing the information density of transferred data. In distributed database systems, compressed data can be transferred faster across the network than uncompressed data. Uncompressed data require either more network time or a separate compression step. Finally, retaining data in compressed form in the I/O buffer allows more records to remain on the buffer, thus increasing the buffer hit rate and reducing the number of I/Os.

As regards transaction processing, the buffer hit rate should increase since more records fit into the buffer space. Since the log records can become shorter, I/O to log devices should decrease. It is trivial to include compressed values if the log of the database contains them. It means, compressing the database values also reduces the size of log records, and therefore the I/O traffic to log devices. It improves the I/O performance on both the primary database and the log [4].

The compression of the inserted data is shown in Figure 8.19. The block is empty at the beginning. When data is inserted into blocks, it is stored in uncompressed format. After filling the block, the compression is triggered, and the block has compressed data and free space for another inserted data. Then again uncompressed data can be loaded until the block is filled and triggers data compression.

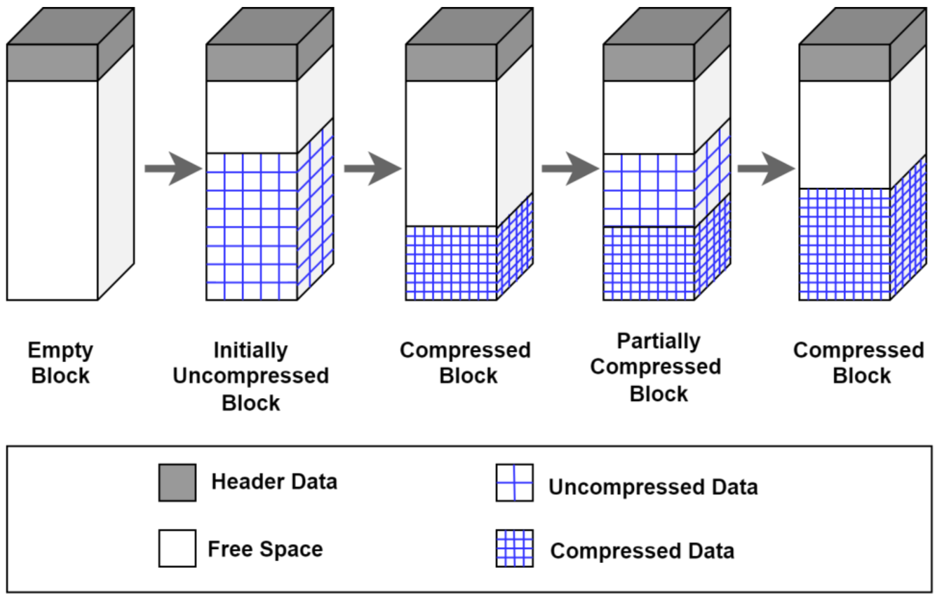


Fig. 8.19. Compression steps

8.5.2 COMPRESSED VS. UNCOMPRESSED BLOCK

Compressed database blocks are very similar to normal uncompressed blocks. Code modifications done in the Oracle RDBMS server to allow for compression are very localized. The only edited parts of code were the ones with formatting blocks and accessing rows and columns. As a result, accessing a compressed block is completely transparent to the database user or any application. Except for dropping columns, there is no other limitation, and all other database features and functions that work on regular database blocks also work on compressed database blocks [22] [77].

Figure 8.20 shows an uncompressed and compressed block with the data listed in the top part. In an uncompressed block, all the redundant information is stored. In a compressed block, instead of storing all data, redundant information is replaced by links to a common reference in the symbol table, indicated by the black dots. This is referred to as cross-column compression. Only entire column values or sequences are compressed. Sequences of columns are compressed as one entity if a sequence of column values occurs multiple times in many rows.

RowId	Invoice_ID	C_First_Name	C_Name	Sales_amt
1	124301	Henry	McGee	13.99
2	992303	Henry	Todd	1.99
3	983302	Todd	McGee	1.99
4	123303	Tom	Smith	1.99
5	121230	Tom	Smith	1.99
6	213305	Henry-Todd	McGee	1.99

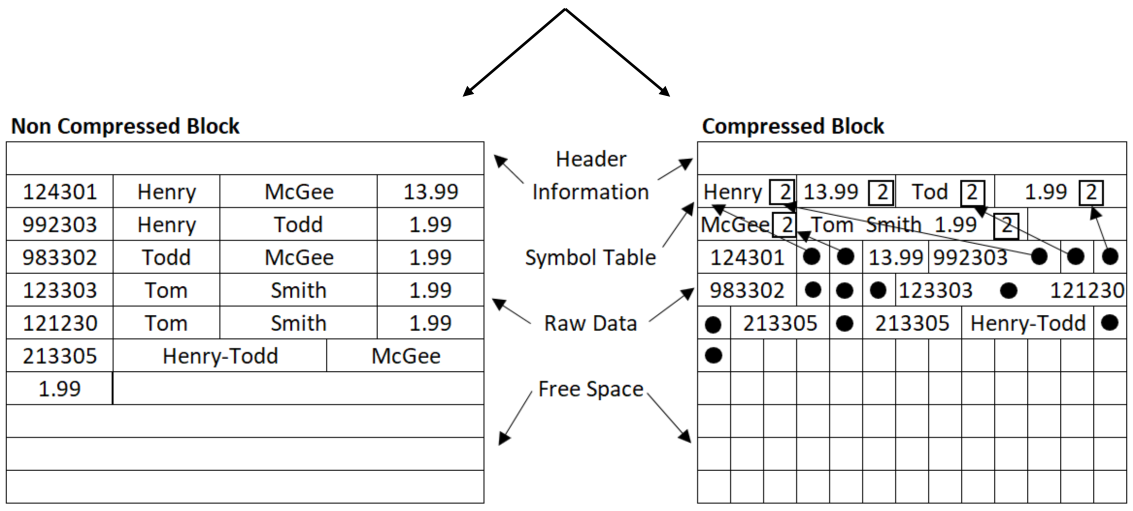


Fig. 8.20. Difference between uncompressed and compressed blocks

8.5.3 COMPRESSED TABLE

Table compression can reduce disk and buffer cache requirements for database tables. Since the compression algorithm utilizes data redundancy to compress data at a block level, the higher the data redundancy is within one block, the larger the benefits of compression are. If a table is defined as compressed it will use fewer data blocks on disk, thereby, reducing disk space requirements [59]. Data from a compressed table is read and cached in its compressed format and it is decompressed only at data access time. Because data is cached in its compressed form, significantly more data can fit into the same amount of buffer cache as Figure 8.21 shows.

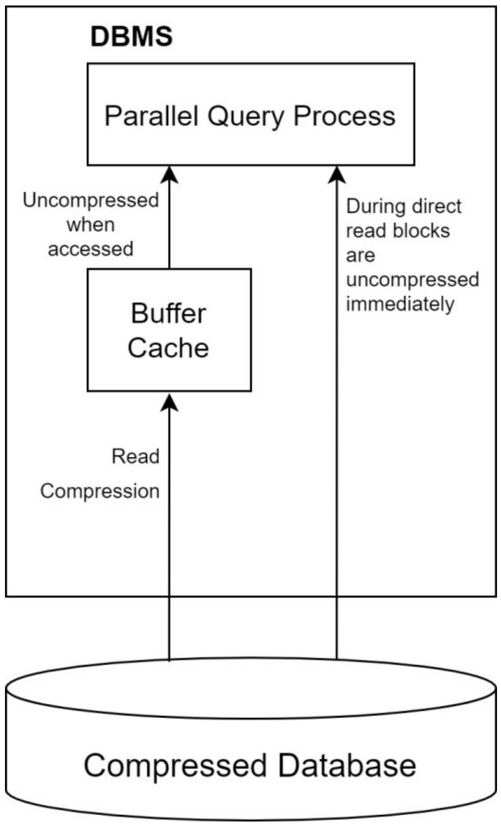

DBMS
Parallel Query Process
Uncompressed when accessed
During direct read blocks are uncompressed immediately
Buffer Cache
Read Compression
Compressed Database

Fig. 8.21. Data access path with compression

Table compression is enabled by adding the *COMPRESS* keyword to the end of the table definition, as follows:

```
CREATE TABLE table_1 (attribute_1 data_type_1 NOT NULL,
                      attribute_2 data_type_2 NOT NULL
                      ) COMPRESS;
```

The default compression status when no compression clause is specified is *NOCOMPRESS,* and it can be displayed using the *COMPRESSION* column in the DBA_TABLES, ALL_TABLES, or USER_TABLES views. The column called *COMPRESS_FOR* indicates the type of compression. It is displayed as follows:

```
SELECT compression, compress_for
  FROM user_tables
     WHERE table_name = 'TABLE_1';
```

The results of the stated select statement showed the value of the attribute *COMPRESS* as *enabled* and the value of the attribute *COMPRESS_FOR* as *basic*. The compression status can be changed by altering a table and defining the new status. However, it will not affect the existing data. It will affect the compression of new data that is loaded by direct path loads. To affect the compression of already existing data, it is necessary to perform a move operation, so the data gets compressed or uncompressed during the copy, as follows:

```
ALTER TABLE table_1 MOVE NOCOMPRESS;
```

However, performing a move operation keeps a new copy of the table for a period of time, which can often be disadvantageous in terms of costs.

8.5.4 METHODOLOGY

Data access time characteristics and CPU costs have been obtained by using the Oracle 18c database system based on the relational platform.

Experiment results were provided using Oracle Database 18c Express Edition Release 18.0.0.0.0 – Production Version 18.4.0.0.0. Parameters of the used computer are:

- Processor: Intel(R) Core(TM) i5-3317U; 1.70GHz,
- Operation memory: 8GB,
- HDD: 500GB.

To compare the data access time and CPU costs, four tables were created over which selecting data was performed. These tables had identical four columns of *Integer*, *Varchar2*, and *Date* data types. Two tables were created with no compression and two tables were compressed. The tables contained one, and two million rows. In addition to ID, the other attributes contained non-unique values. Compressed tables were created as follows:

```
CREATE TABLE tableA (id INTEGER PRIMARY KEY,
                     name VARCHAR2(30),
                     dateA DATE,
                     amount INTEGER) COMPRESS;
```

Uncompressed tables were created in the same way but without the *COMPRESS* keyword. Characteristics of created scenarios are shown in Table 8.5.

	NAME	**MILLION ROWS**	**TABLE COMPRESSION**	**INDEX COMPRESSION**
1	tab1_NO	1	X	X
2	tab1_NO	1	X	✓
3	tab2_COM	1	✓	X
4	tab2_COM	1	✓	✓
5	tab3_NO	2	X	X
6	tab3_NO	2	X	✓
7	tab4_COM	2	✓	X
8	tab4_COM	2	✓	✓

Tab. 8.5: Created scenarios

Compressed indexes were created as follows:

```
CREATE INDEX index1_com
  ON tab2_COM(name) COMPRESS;
```

Uncompressed indexes were created in the same way but without the *COMPRESS* keyword. Select statements were executed over the created tables, which resulted in 38 596 rows out of 1 million rows and in 77 192 rows out of 2 million rows. The type of select statement was as follows:

```
SELECT count(*)
  FROM table1_NO
    WHERE name like 'M%';
```

8.5.5 RESULTS

Data access times of executed scenarios are shown in Table 8.6 in milliseconds.

NUMBER OF INDEXES	**INDEX COMPRESSION**	**tab1_NO** 1 million rows	**tab2_COM** 1 million rows	**tab3_NO** 2 million rows	**tab4_COM** 2 million rows
0	-	138	143	264	274
1	✓	27	24	38	38
2	✓	24	24	36	37
3	✓	24	26	37	37
1	X	25	22	32	32
2	X	24	22	32	34
3	X	24	22	33	34

Tab. 8.6: Data access times in milliseconds

Based on the results, it is clear that data compression affects data access time. The use of an index had a significant reduction in data access time, reducing the time to about 13 to 19% of the total data access time during no index use. The addition of other indexes has brought only minor changes. Situations using an uncompressed index generally performed better, but the difference was not very large. For a table with 1 million rows, it was more advantageous to use compression over the table. However, for a table with 2 million rows, compressing the table did not result in lower data access times.

Another observed phenomenon was the CPU cost, which is shown in Table 8.7.

NUMBER OF INDEXES	**INDEX COMPRESSION**	**tab1_NO** 1 million rows	**tab2_COM** 1 million rows	**tab3_NO** 2 million rows	**tab4_COM** 2 million rows
0	-	1071 (2)	969 (2)	2152 (1)	1974 (2)
1	✓	122 (0)	108 (0)	46 (0)	177 (1)
2	✓	122 (0)	108 (0)	46 (0)	177 (1)
3	✓	122 (0)	108 (0)	46 (0)	177 (1)
1	X	92 (0)	82 (0)	48 (0)	182 (1)
2	X	92 (0)	82 (0)	48 (0)	182 (1)
3	X	92 (0)	82 (0)	48 (0)	182 (1)

Tab. 8.7: Cost (%CPU)

The positive effect of compression was much more noticeable when observing CPU costs. For a table with 1 million data, the cost of using table compression was reduced from 122 to 108 when using a compressed index and from 92 to 82 when using an uncompressed index. For a table with 2 million data, however, the compression of the table brought worse costs, but on the contrary, the compression of the index reduced the total costs from 48 to 46 and from 182 to 177.

Both data access times and CPU costs in a non-indexed situation were too high due to a full table scan. When using the index, the index range scan method was performed.

8.5.6 SUMMARY

Performance of the data management in the database is a crucial element of almost any information system. It is evident, that the amount of data to be processed and evaluated is still rising very rapidly. The impact of index and table compression with non-unique data on data access time and CPU costs was observed. Eight different scenarios were compared, which differed in the status of compression of a table, or an index created over the table, and also in the number of table rows.

The attributes of the tables contained non-unique values, and thus the table had a low cardinality, the effect of compression was bigger than in the previous experiments with unique values. The positive effect of compression was visible in some situations, especially when using table compression with 1 million rows. In addition to data access times, the impact was even more visible when comparing CPU costs. However, this effect varied with the number of rows in the tables and with the compression of the index.

8.6 THE COMPLEXITY OF THE DATA RETRIEVAL PROCESS USING THE INDEX EXTENSION

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 [42]. 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. 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. 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.

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. By reaching the complexity of the proposed architecture, significant performance improvements can be identified.

8.6.1 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, which can 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, to locate the row using the index, multiple data blocks need to be loaded. 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 not suitable, and it is necessary to apply and limit migrations online dynamically. To analyze the impact and evaluate benefits, we propose and discuss different solutions. 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.

8.6.1.1 MIGRATED ROWS – SOLUTION 1

The first solution limiting the impact of migrations is based on the data block structure extension. For each data row, a list of the pointers to the index set is present. 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 currently resides. From the physical point of view, for each row, a new dynamic array is allocated inside the main block, which is shown in Figure 8.22. As evident, if a new index is created, a new array element must be used to reference the index.

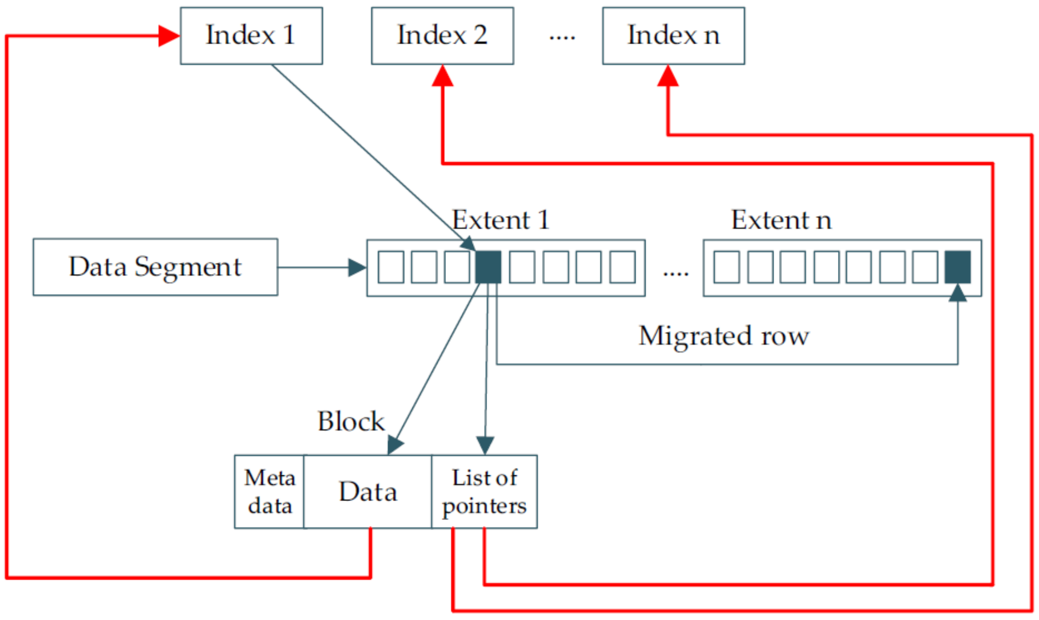


Fig. 8.22: Architecture of solution 1 – list of pointer structure

Therefore, each operation is applied at the index level, but also 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 was set as *NOLOGGING* after the instance failure, it would be necessary to reconstruct the entire index, and also 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, as well. 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, if the number of indexes was too large, 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).

8.6.1.2 MIGRATED ROWS – SOLUTION 2

The first solution 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 the searching can be done more efficiently. Each element has an object identifier, extended by the physical data position (ROWID), and a reference list to the indexes. Such a 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. Figure 8.23 shows the *SOLUTION 2a* architecture.

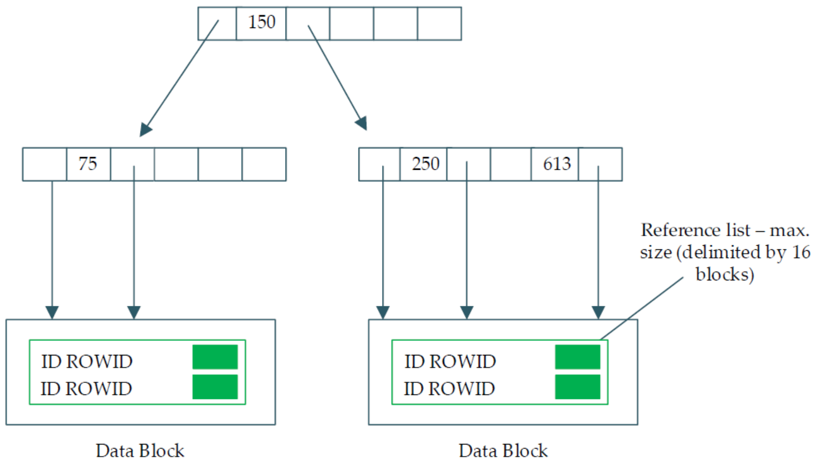


Fig. 8.23: Architecture of solution 2a – inline reference list

The architecture of SOLUTION 2b is shown in Figure 8.24. The overflow structure is optional and added dynamically if one block cannot cover the reference list completely. 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.

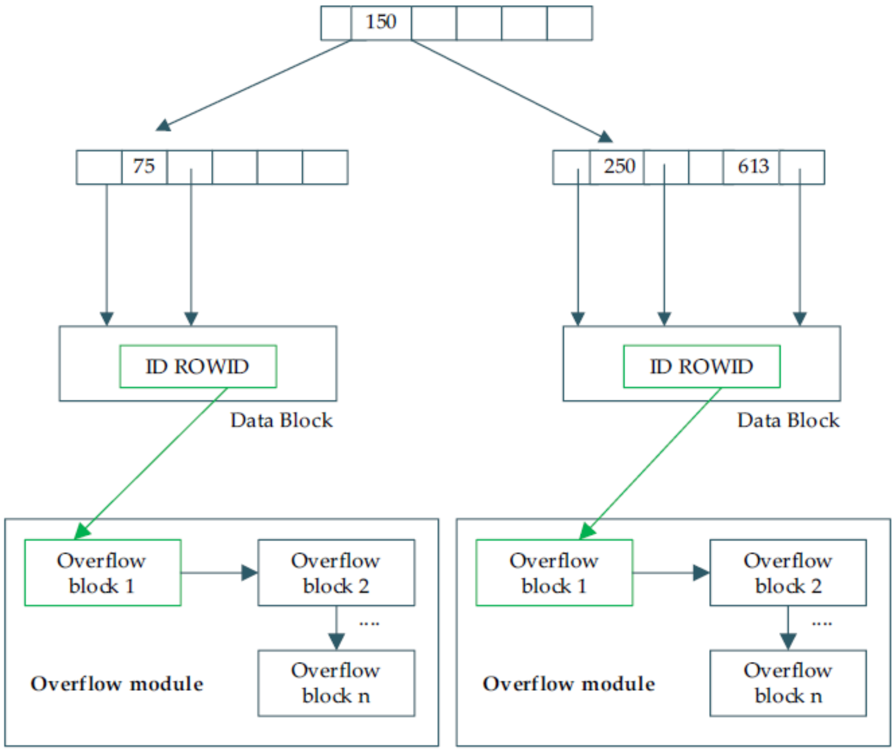


Fig. 8.24: Architecture of solution 2b

8.6.2 MIGRATION-POST-APPLICATION

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

The solution is based on the identification of 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 disadvantage of the solution is represented by the I/O operation extension. *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 of the index access. The solution of the migration mapper is shown in Figure 8.25. There can be several levels of migration for a particular row. Each index can reference a different root node of the *Migration_mapper*. Thus, the reference model cannot be directly cascaded.

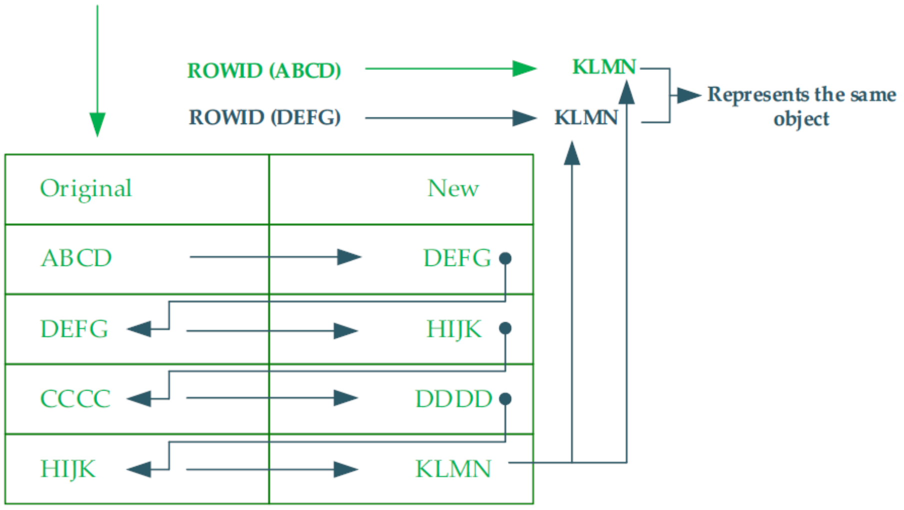


Fig. 8.25: Architecture of solution 3 – Migration mapper

8.6.3 STORING PATH

Another concept is covered by the fourth solution. Although the connection between the index and data is just one-directional, during the access using the defined index, the traverse path can be temporarily stored, thus if the data migration is detected, by such definition, the 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. The 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 data reflector structure operated by the Session-Reflector background process 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 to 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.

8.6.4 DATA POINTER REFLECTION

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). The Data pointer reflector is the index submapper structure transforming the 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 in 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. The 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.

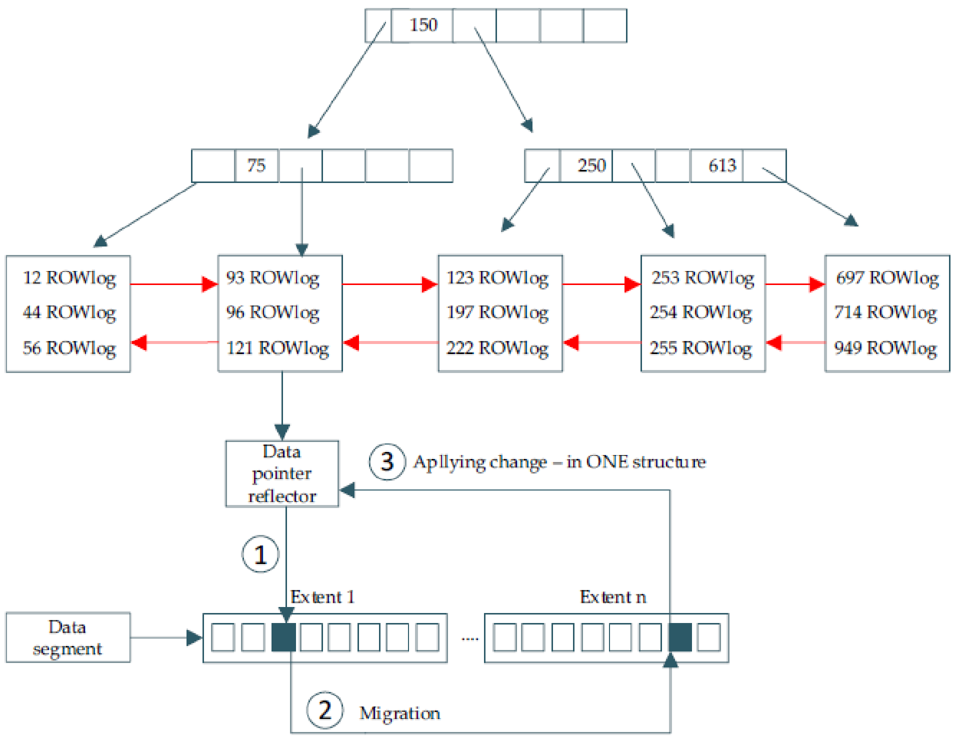


Fig. 8.26: Architecture of solution 5 – Data pointer reflector

The logical model of the Data pointer reflector is shown in Figure 8.26. 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 one reference needs to be updated, and the whole index set can remain original. Each index is routed to the same Data pointer reflector associated with the particular table.

8.6.5 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 in 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. Our research emphasis 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.

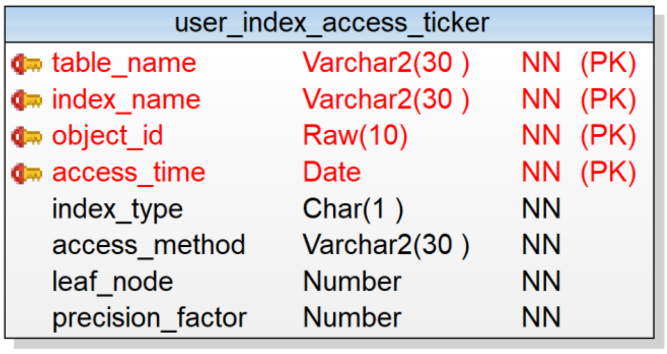
For each data row of the registered table, the index is extended by evaluating priority, calculated by the frequency of access. It is handled in the index granularity, whereas the tuple itself is composed of attributes with various frequencies, precision, and durability of the update and Select statement. 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 for managing the various amounts of index-covered data, it is evident that it is rather broad than deep. In chapter 4.1, there is a table 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 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, shown also in Figure 8.27:

- table_name – the name of the 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 – the rebalancing precision factor of the index (used for the index optimization complexity calculation).



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	

Fig. 8.27: Access Ticker data dictionary structure

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, the 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 on 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. Figure 8.28 shows the current data flow.

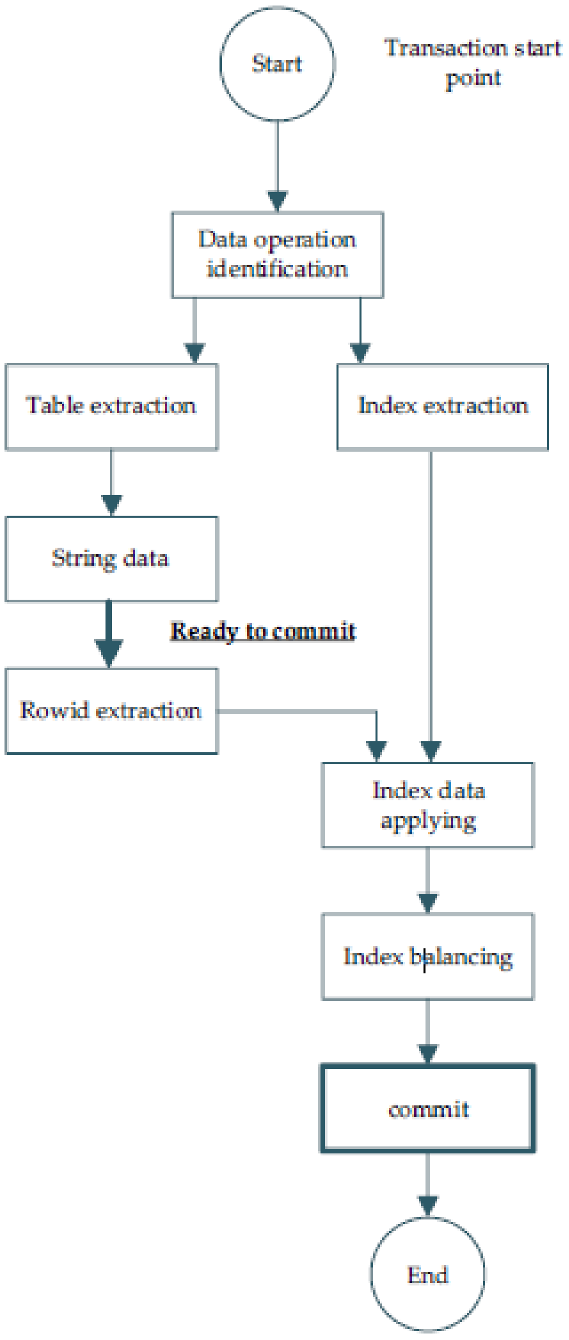


Fig. 8.28: Existing data flow using the index balancing

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, and 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 the 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. Figure 8.29 shows the architecture and data flow of the proposed solution.

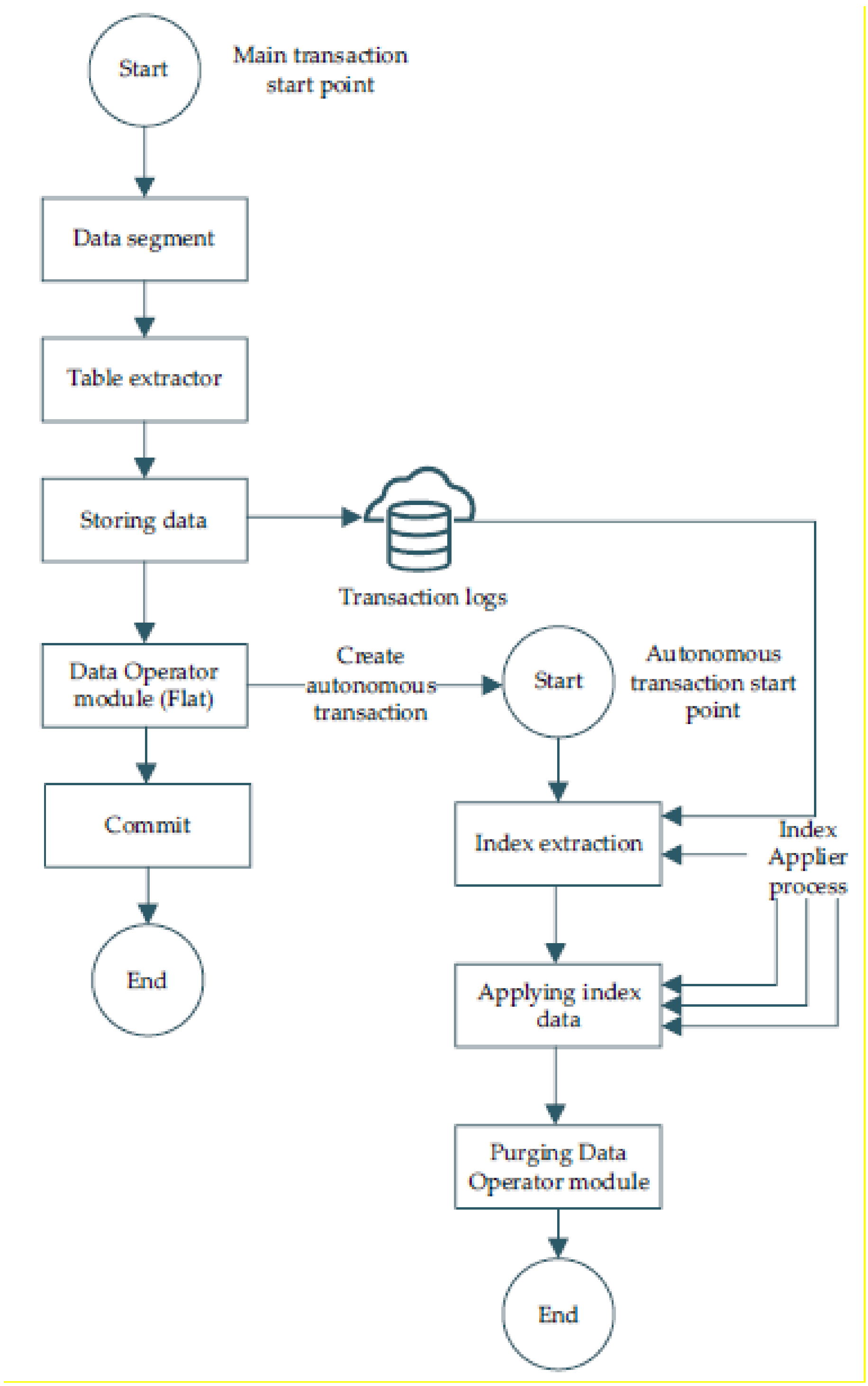

Start
Main transaction start point
Data segment
Table extractor
Storing data
Transaction logs
Data Operator module (Flat)
Create autonomous transaction
Start
Autonomous transaction start point
Commit
End
Index extraction
Index Applier process
Applying index data
Purging Data Operator module
End

Fig. 8.29: 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 number 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.

8.6.6 ARCHITECTURE ENHANCEMENTS

Currently, the most often used type is the 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 [47], there were 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 [62] 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.

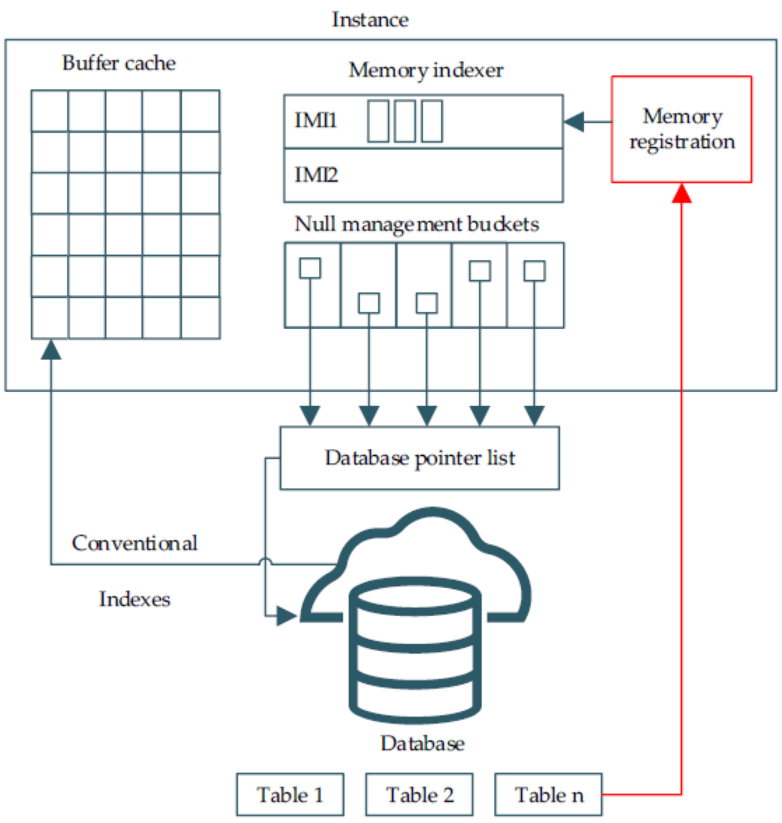


Fig. 8.30: Architecture – memory registration

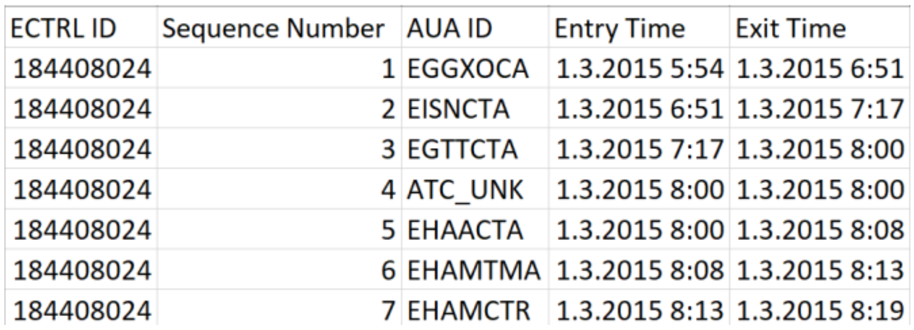
We use another perspective reflecting the memory. Instead of using the index as a physical database segment, the proposed solution locates the column store in the memory, as shown in Figure 8.30. It treats individual attributes separately, even for composite indexes. Attribute column store structure 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.

8.6.7 METHODOLOGY

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 spatiotemporal 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:

- 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.

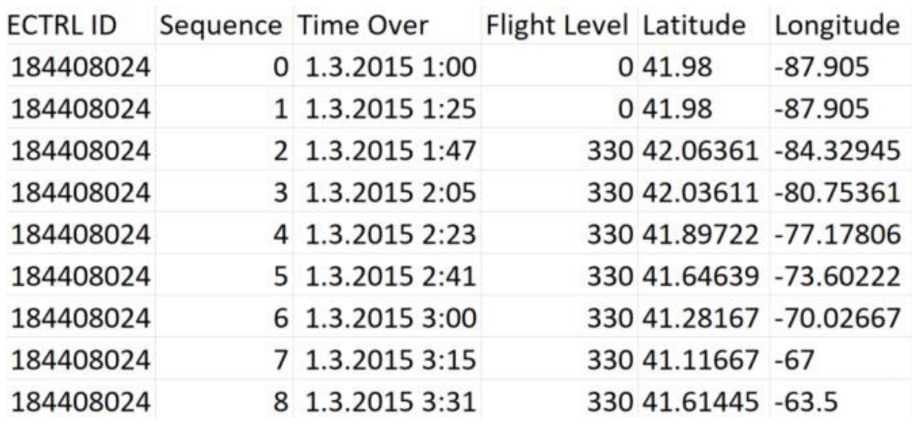
Figure 8.31 shows the data example of the flight area - airspace (FIR) assignment. There is an object identifier (ECTRL ID) – airplane license plate, sequence number related to the specific flight ordering the data. AUA value identifies the flight space assignment, which can evolve over time, as well. 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.



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

Fig. 8.31: Data – Airspace assignment

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, as shown in Figure 8.32. Finally, 0.5 GB of storage was used for indexes and structural extensions.



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

Fig. 8.32: 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, a 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 the 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. 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) for one day (24 hours), for the workday and weekend separately,
    - monitoring the airplane during the whole flight (flight time in the range of one to two hours),
    - monitoring the airport for one day,
- data loading and change process.

8.6.8 RESULTS

8.6.8.1 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. 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, and 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 a 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. A 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. The 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.

In 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 to 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 Table 8.8. 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.

Performance results - additional costs of the migration management			**Absolute values for referential model**	Reference model No migration management	Extension requirements						
					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

Tab. 8.8: Results – Extension Requirement for Data Migration Management

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 the proposed SOL_MIG_5 is more significant. Namely, table 8.9 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 ensure the performance of the degraded physical architecture. For declarative purposes, values are expressed in percentages.

Data retrieval - Extension requirements				
		Reference model No migration management	SOL_MIG 5b	Difference
		%	%	%
migrated row ratio	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

Tab. 8.9: Results – Impact of Data Migration Percentage

Figure 8.33 shows the results in a graphical form.

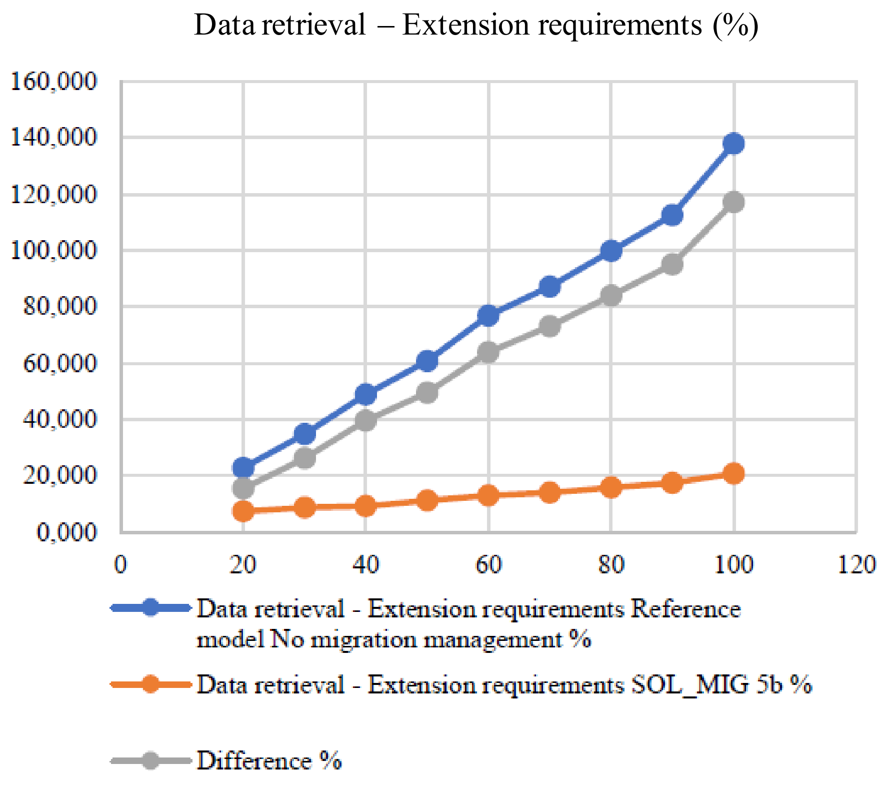


Fig. 8.33: Results – Impact of SQL_MIG_5b on the performance

8.6.8.2 PRIORITY MANAGEMENT

In the real-time control and safety systems, it is inevitable to minimize the time consumption to obtain relevant data. We have dealt with our own solution by covering priority in the B+ tree structure extension. This evaluation discusses the impact and benefits of such a model in 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 the airplane during the whole flight (flight time in the range of one to two hours),
    - monitoring the airport for 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 8.10 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, and additional 32 extends 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.

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

Tab. 8.10: Priority Management – Size Demands

Table 8.10 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 8.11 using the second precision. As can be seen from the results, there is just a slight performance benefit for the whole airspace monitoring. Without any priority management, the total time demands are 4.194 seconds. By using the 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, an even less significant difference expressed in percentage is present – no more than 0.6% at the microsecond level.

Similarly, monitoring the airport for 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_PRIORITY_2). 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. If GB or TB of data is 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.

Name	SOL_PRIORITY_REF	SOL_PRIORITY_1	SOL_PRIORITY_2
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 for one day (seconds)	1.823	1.805	1.784

Tab. 8.11: Priority Management – Time Processing

Table 8.12 shows the demands on the rebalancing in time representation (second precision).

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

Tab. 8.12: Priority Management - Rebuilding

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 specific 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. Figure 8.34 shows the results in the graphical form.

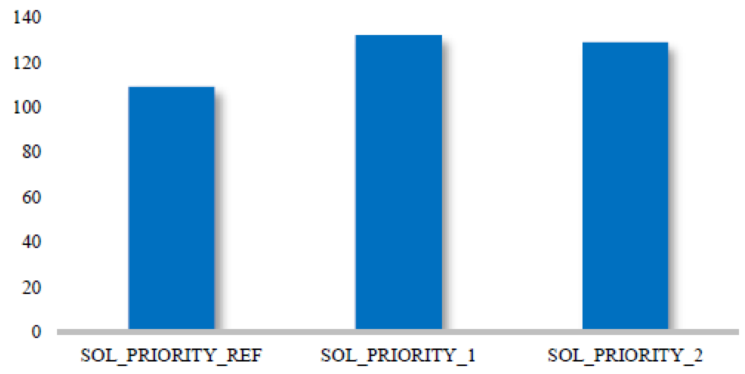


Fig. 8.34: Rebuilding operation demands – priority management results (seconds)

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 8.13 shows the results for 100 000 balancing operations based on the same environment.

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

Tab. 8.13: Rebalancing Time – Priority Management

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 the 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.

8.6.8.3 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). A performance study is done for the data retrieval and inserting of a new state into the database.

Table 8.14 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. 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.

		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

Tab. 8.14: Data Retrieval Process

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%.

8.6.8.4 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 in 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.

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 nontrusted 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 8.15. 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%.

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

Tab. 8.15: Undefined Value Management – Results

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.

8.6.9 SUMMARY

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. 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 to 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.

We have dealt with index management by extending the structure. The common index type is the 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 of grouping multiple states to be applied at once. Thanks to the proposed architecture, data can be processed and evaluated sooner, and the index set is reliable, covering all the data with the extensional modules.

Fragmentation is just one element of the physical architecture in terms of blocks. 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. We have proposed 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.

9 CONCLUSION

In the dissertation thesis, we have dealt with indexing, deployment, and searching algorithms in large databases. The first part of the work, containing chapters 2 to 7, can be considered a theoretical and partly practical part. It contains an introduction to the issues of individual areas and an analysis of the possibilities of their application. In the second chapter, we dealt with the types of database systems, whether distributed, relational or object-relational. We also focused on data warehouses and data marts and on their comparison, either with each other or with a classical database. In the third chapter, we dealt with searching in data structures. We have described different types of searches and different structures that are primarily intended for search but are not entirely suitable for use in database systems because they do not have the best time to insert and delete data. The fourth chapter was focused on indexing. It contains an analysis of several types of indexes and methods used to scan indexes. In this chapter, we also focused on the strategies of performing the join operation, the method of reclaiming unused disks using shrinking the space, as well as the method of full-text search and automatic indexing. In the fifth chapter, we dealt with partitioning. This chapter contains an analysis of three techniques for creating partitions, which are then used in various partitioning methods, and also a method of ensuring that logical partitions are mapped to physical groups of files, based on a partitioning scheme. Attention is also paid to the partitioned index, whether locally or globally partitioned. The sixth chapter focuses on automatic storage management, which simplifies the management of files, control files, and log files. In the seventh chapter, we analyze the institutes and universities that deal with similar issues and emphasize some of their research.

The second part of the work is contained in the eighth chapter and contains our own contribution based on various performed experiments. In chapter 8.1, we dealt with the master index, where the aim was to limit the necessity of using sequential data block scanning performed by the Table Access Full method. The master index is not used for the evaluation itself, whereas it does not fit the conditions of the query. The core is that it contains all the pointers to the data on the leaf layer, so the index itself is used as the data locator. In chapter 8.2, we focused on reducing data access time using various partitioning techniques and methods. We executed several scenarios differing in types and numbers of created partitions. The results of the experiments showed that access to data stored in partitions can significantly reduce the time of access to data and thus bring greater efficiency. Chapter 8.3 deals with the effect of partitioning and indexing. The access times for the data in the table with the created partitions and the table without any partitions were compared. Various situations differed also in the type of indexes created on both tables, such as non-partitioned, global partitioned, local partitioned prefix, and local partitioned non-prefixed indexes. From the results, it was concluded, that the use of partitioning, indexes, or their combinations can significantly speed up data access time. However, with frequent access to a large amount of data to which a large number of partitions or index nodes are bound, the access time is similar to the scenario without partitions and indexes, or in some situations, it may be even much worse. In chapter 8.4, we dealt with the effect of indexes on DML operations. We performed the experiments on a bike-sharing system. Six scenarios, differing in the number of created indexes related to the modified data, were tested for each of the inserting, updating, and deleting situations. We concluded that the number of indexes makes a significant impact on increasing the time to perform insert, update, and delete operations. A large number of indexes is only suitable for data that does not change often and is only used for retrieval. In situations with frequent and large data changes, it is better to use a smaller number of corresponding indexes. Chapter 8.5 focuses on the impact of table and index compression on data access time and CPU costs. Experiments were executed in eight different scenarios of various combinations of compressed and uncompressed indexes over compressed and uncompressed tables with non-unique data of various row numbers. When the attributes of the tables contained non-unique values, and thus the tables had low cardinality, the effect of compression was more visible than in our previous experiments[6] with tables containing unique data. Chapter 8.6 deals with the complexity of the data retrieval process using the index extension. Here we have focused on migrated rows identification and reflection, index automatic balancing, its structure efficiency evaluation, and management outside the main transaction. We have also dealt with the control of undefined values, relevant data block identification, and dynamic execution plan optimization.

Our experiments were performed in the Oracle relational database system and were also tested in the MySQL relational database system. However, some parts of the solutions and scenarios have undergone minor changes, with other systems addressing the available functionalities differently from Oracle or not offering them at all. This was the case, for

[6] ŠALGOVÁ, V., KVET, M. (2021). *The Impact of Table and Index Compression.* 19th International Conference of Emerging eLearning Technologies and Applications (ICETA).

example, when creating index partitions over a non-partitioned table, or vice versa when creating an unpartitioned index over a partitioned table. Such a solution in Oracle is possible, but MySQL does not allow it. Another example of a different solution across other database systems is with undefined values. A NULL value works in Oracle as an empty string, while in MySQL after concatenating a string and a NULL value we get a NULL value. To achieve an analogous solution, it would be necessary to apply a transformation function.

In general, these solutions are universal and applicable in a variety of systems. Our methods are applicable to any structure that can be indexed. In situations that are enriched with collections that cannot be indexed, it is possible to use the *TABLE* keyword to access the collection as a relational table and then index it. So, there would be a need for an intermediate layer that would transform collections into relational tables, create indexes, and then make a collection out of it again. When implementing XML type into the solution, it is possible to index their individual elements. They correspond to object-relational columns and tables. Creating B-tree indexes on those columns and tables thus provides an excellent way to effectively index the corresponding XML objects. What is more, XMLIndex provides a general, XML-specific index that indexes the internal structure of XML data. One of its main purposes is to overcome the indexing limitation presented by binary XML storage.

As for JSON data type, there is no dedicated SQL data type for JSON data, so it can be indexed in the usual ways. It is possible to define a JSON search index, which is useful for both ad hoc structural queries and full-text queries. In particular, a B-tree index or a bitmap index for SQL/JSON function *json_value* can be used. Such function-based indexing is appropriate for queries that target particular functions, which in the context of SQL/JSON functions means particular SQL/JSON path expressions. Both function-based indexes and a JSON search index can be defined for the same JSON column.

Through our experiments and performance evaluation study, we have developed a methodology for increasing the efficiency and effectiveness of data access, mainly through the use of indexes and their automatic balancing, partitioning, or a combination thereof, the effect of data compression or dynamic execution plan optimization. Using the master index as the data locator can limit the necessity of using sequential data block scanning performed by the Table Access Full method. Partitioning of tables and indexes can significantly reduce the time of access to data and thus bring greater efficiency. However, with frequent access to a large number of partitions or index nodes, or with frequent changes of data it is not always suitable to use partitioning or indexing. The number of indexes makes a significant impact on increasing the time to perform insert, update and delete operations. A large number of indexes is only suitable for data that does not change often and is only used for retrieval. In situations with frequent and large data changes, it is better to use a smaller number of corresponding indexes. Using compression of data is convenient with non-unique values, in a comparison with unique data when the improvement is less visible. Index management by extending the structure uses Data Indexers to exclude balancing operations inside the transaction, so the original transaction can be approved sooner and the balancing benefits of grouping multiple states to be applied at once. Data can be processed and evaluated sooner, and the index set is reliable, covering all the data with the extensional modules. Dealing with fragmentation and migrated rows, we have proposed new structures and background processes responsible for identifying migrated rows and applying changes. 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. such a mapping structure can bring significant performance benefits, whereas there is just one place to place and locate migration. Total costs and processing time are improved, even when the additional data structure is used.

In the near future, we would like to focus on dynamic data balancing in created partitions, as well as situations with hybrid partitioning. An interesting topic of our research could be also the multi-index, in which both indexes could be processed in parallel and possibly compressed. We would like to deal also with the dependence of performance on the structure and hierarchy of tables and the relationships between them, and thus on the effect of join levels on execution time. We would also focus on the impact of block sizes and disk types on which partitions are created, on performance, and data access times.

LIST OF OWN PUBLICATIONS

[I] ŠALGOVÁ, V., KVET, M. (2019). *Optimization Methods in Bike-Sharing Systems.* 17th International Conference on Emerging eLearning Technologies and Applications (ICETA). - pp. 687-692 - doi: 10.1109/ICETA48886.2019.9040079.

[II] ŠALGOVÁ, V., KVET, M. (2019). *Intelligent transport and parking systems.* Dopravná infraštruktúra v mestách: zborník príspevkov z 10. Medzinárodnej konferencie. – 1.vyd. – Žilina: Žilinská univerzita v Žiline. – pp. 1-8. - ISBN: 978-80-554-1594-9.

[III] KVET, M., ŠALGOVÁ, V., KVET, M., MATIAŠKO, K. (2019). *Master Index Access as a Data Tuple and Block Locator.* Proceedings of the 25th Conference of Open Innovations Association (FRUCT). – pp. 176-183 – ISBN: 978-952-69244-1-0 - doi. 10.23919/FRUCT48121.2019.8981531.

[IV] ŠALGOVÁ, V., MATIAŠKO, K. (2020). *Analysis of Very Large Database Indexing.* International Scientific Days 2020: Innovative Approaches for Sustainable Agriculture and Food Systems Development. – ISBN: 978-963-269-918-9.

[V] ŠALGOVÁ, V., MATIAŠKO, K. (2020). *Reducing Data Access Time using Table Partitioning Techniques.* 18th International Conference of Emerging eLearning Technologies and Applications (ICETA). – pp. 564-569 – ISBN: 978-0-7381-2366-0.

[VI] ŠALGOVÁ, V., MATIAŠKO, K. (2021). *The Effect of Partitioning and Indexing on Data Access Time.* Proceedings of the 29th Conference of Open Innovations Association (FRUCT). – pp. 301-306 – ISBN: 978-952-69244-5-8 - doi: 10.23919/FRUCT52173.2021.9435500.

[VII] ŠALGOVÁ, V. (2021). *Effect of indexes on DML operations.* 14th International Scientific Conference on Sustainable, Modern and Safe Transport (TRANSCOM). – pp. 1368-1372 – ISSN 2352-1465 - Code: 146198.

[VIII] KVET, M., ČEREŠŇÁK, R., ŠALGOVÁ, V. (2021). *Use of Machine Learning for the Unknown Values in Database Transformation Processes.* 11th International Scientific Conference on Communication and Information Technologies (KIT). – pp. 1-7 – doi: 10.1109/KIT52904.2021.9583753.

[IX] ŠALGOVÁ, V., KVET, M. (2021). *The Impact of Table and Index Compression.* 19th International Conference of Emerging eLearning Technologies and Applications (ICETA) - pp. 327-332, doi: 10.1109/ICETA54173.2021.9726601.

[X] ŠALGOVÁ, V. (2022). *The Impact of Table and Index Compression on Data Access Time and CPU Costs.* 10th World Conference on Information Systems and Technologies (WorldCIST).

REFERENCES

[1] ABDELHAFIZ, B.M. (2020). *Distributed Database Using Sharding Database Architecture*. IEEE Asia-Pacific Conference on Computer Science and Data Engineering, CSDE 2020. doi: 10.1109/CSDE50874.2020.9411547.

[2] ADAIR, B. (2019). *BI/DW: What is Business Intelligence and Data Warehousing?* Retrieved from: https://www.selecthub.com/business-intelligence/business-intelligence-and-data-warehousing/.

[3] ADAMU, F. B., HABBAL, A., HASSAN, S., COTTRELL, R. L., WHITE, B., ABDULLAHI, I. (2016). *A Survey on Big Data Indexing Strategies*. The 4th International Conference on Internet Applications, Protocols and Services.

[4] AGHAV, S. (2010). *Database compression techniques for performance optimization*. 2nd International Conference on Computer Engineering and Technology. IEEE. p. V6-714-V6-717.

[5] AHSAN, K., VIJAY, P. (2014). *Temporal Databases: Information Systems.* Booktango.

[6] AL-SANHANI, A.H. et all. (2017). *A comparative analysis of data fragmentation in distributed database*. ICIT 2017 - 8th International Conference on Information Technology, Proceedings. pp. 724–729. doi: 10.1109/ICITECH.2017.8079934.

[7] ARORA, G., KALRA, S., BHATIA, A., TIWARI, K. (2021). *PalmHashNet: Palmprint Hashing Network for Indexing Large Databases to Boost Identification*. IEEE Access. 9, pp. 145912–145928. doi: 10.1109/ACCESS.2021.3123291.

[8] ASHDOWN, L., KYTE, T. (2015). *Oracle database concepts.* Oracle Press.

[9] BURLESON, D.K. (2001). *Oracle high-performance SQL tuning*. McGraw-Hill, Inc.

[10] CAMBAZOGLU, B. et all. (2013). *A term-based inverted index partitioning model for efficient distributed query processing*. ACM Transactions on the Web, Volume 7, Issue 3. pp 1-23. doi: 10.1145/2516633.2516637.

[11] CERI, S., NEGRI, M., PELAGATTI, G. (1982). *Horizontal data partitioning in database design*. ACM SIGMOD International Conference on Management of Data. pp. 128-136.

[12] CHAN, H.L., HON, W.K., LAM, T.W. (2004). *Compressed index for a dynamic collection of texts*. In*: Annual Symposium on Combinatorial Pattern Matching.* Springer, Berlin, Heidelberg. p. 445-456.

[13] CHEN, H., LI, J. (2013). *The Research of Embedded Database Hybrid Indexing Mechanism Based on Dynamic Hashing*. Lecture Notes in Electrical Engineering. 211 LNEE, pp. 691–697. doi: 10.1007/978-3-642-34522-7_74.

[14] CHENG, C.H., WEI, L.Y., LIN, T.C. (2007). *Improving relational database quality based on adaptive learning method for estimating null value*. Second International Conference on Innovative Computing, Information and Control, ICICIC 2007. doi: 10.1109/ICICIC.2007.350.

[15] CORNEJO, R. (2018). *Dynamic Oracle Performance Analytics*. doi: 10.1007/978-1-4842-4137-0.

[16] DARGAHI NOBARI, A., RAFIEI, D. (2021). Efficiently Transforming Tables for Joinability. doi: arxiv-2111.09912.

[17] DATE, C.J., LORENTZOS, N., DARWEN, H. (2015). *Time and Relational Theory: Temporal Databases in the Relational Model and SQL.* Morgan Kaufmann.

[18] DELPLANQUE, J., ETIEN, A., ANQUETIL, N., AUVERLOT, O. (2018). *Relational database schema evolution: An industrial case study.* IEEE International Conference on Software Maintenance and Evolution ICSME 2018 - pp. 635-644.

[19] DESAI, M., MEHTA, R., RANA, D. (2019). *A Survey on Techniques for Indexing and Hashing in Big Data*". doi: 10.1109/CCAA.2018.8777454.

[20] ERLANDSSON, M. et all. (2016) *Spatial and temporal variations of base cation release from chemical weathering a hisscope scale*. In Chemical Geology, Vol. 441, pp. 1-13.

[21] EZEIFE, C.I., ZHENG, J. (1999). *Measuring the performance of database object horizontal fragmentation schemes*. Proceedings of the International Database Engineering and Applications Symposium, IDEAS. pp. 408–414. doi: 10.1109/IDEAS.1999.787292.

[22] FERRAGINA, P., MANZINI, G. (2001). *An experimental study of a compressed index*. Information Sciences. 135.1-2: 13-28.

[23] FEUERSTEIN, S. (2007). *Oracle PL/SQL Best Practices: Write the Best PL/SQL Code of Your Life*. O´Reilly Media. Inc. 978-0596514105.

[24] FINIS, J. et all. (2015). *Indexing highly dynamic hierarchical data*. Proceedings of the VLDB Endowment. 8. 986-997. 10.14778/2794367.2794369.

[25] FREITAG, M. et all. (2020). *Adopting Worst-Case Optimal Joins in Relational Database Systems*. Proceedings of the VLDB Endowment, Volume 13, Issue 12. pp 1891–1904. doi: 10.14778/3407790.3407797.

[26] GARCÍA-MOLINA H., ULLMAN J.D., WIDOM J. (2009). *Database Systems: The Complete Book (Second Edition).* New Jersey.

[27] GRAEFE, G. (2003). *Sorting And Indexing With Partitioned B-Trees. CIDR*. Vol. 3.2.

[28] GRAEFE, G., GUY, W., SAUER, C. (2016). *Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, Media Restore, and System Failover, Second Edition*. Synthesis Lectures on Data Management. 8, pp. 1–113.

[29] HAN, W.S., LEE, K.H., LEE, B. (2003). *An XML storage system for object-oriented/object-relational DBMSs. Journal of Object Technology*. 2. 113-126. 10.5381/jot.2003.2.3.a2.

[30] HAJMOOSAEI, A., KASHFI, M., KAILASAM, P. (2011). *Comparison plan for data warehouse system architectures*. The 3rd International Conference on Data Mining and Intelligent Information Technology Applications*, Macao, pp. 290-293.*

[31] HAY, D.C. (2018). *Achieving Buzzword Compliance: Data Architecture Language and Vocabulary*. Technics Publications.

[32] HEMALATHA, G., THANUSKODI, K. (2010). *A Survey on Join Techniques*. Annual International Conference on ADPC. doi: 10.5176/978-981-08-7656-2 A-41.

[33] HONISHI, T., SATOH, T., INOEU, U. (1992). *An index structure for parallel database processing.* Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing.

[34] HUANG, H., LUAN, H. (2021). *Rethinking Insertions to B + -Trees on Coupled CPU-GPU Architectures*. pp. 993–1001. doi: 10.1109/ISPA-BDCLOUD-SOCIALCOM-SUSTAINCOM52081.2021.00139.

[35] INMON, W.H. (1992). *Building the Data Warehouse*. John Wiley & Sons, Inc., USA. ISBN 978-81-265-0645-3.

[36] JIN, D., CHEN, G., HAO, W., BIN, L. (2020). *Whole Database Retrieval Method of General Relational Database Based on Lucene*. Proceedings of 2020 IEEE International Conference on Artificial Intelligence and Computer Applications, ICAICA 2020. pp. 1277–1279. doi: 10.1109/ICAICA50127.2020.9182496.

[37] JOHNSTON, T. (2014). *Bi-temporal data – Theory and Practice.* Morgan Kaufmann.

[38] JOHNSTON, T., WEIS, R. (2010). *Managing Time in Relational Databases.* Morgan Kaufmann.

[39] KAO, M., CERCONE, N., LUK, W.S. (1988). *Providing Quality Responses with Natural Language Interfaces: The Null Value Problem*. IEEE Transactions on Software Engineering. 14, pp. 959–984. doi: 10.1109/32.42738.

[40] KHATRI, H., FAN, J., CHEN, Y., KAMBHAMPATI, S. (2007). *QPIAD: Query processing over incomplete autonomous databases*. Proceedings - International Conference on Data Engineering. pp. 1430–1432. doi: 10.1109/ICDE.2007.369028.

[41] KUHN, D., ALAPATI, S., PADFIELD, B. (2016). *Partitioned Indexes.* In: Expert Oracle Indexing and Access Paths. Apress. Berkeley. CA. doi: 10.1007/978-1-4842-1984-3_6.

[42] KUHN, D., KYTE, T. (2021). *Expert Oracle Database Architecture*. APress. Berkeley. doi: 10.1007/978-1-4842-7499-6.

[43] KUMAR, A. (2018). *Architecting Data-Intensive Applications: Develop scalable, data-intensive, and robust applications the smart way*. Packt Publishing Ltd.

[44] KVET, M. (2021). *Database Index Balancing Strategy*. Conference of Open Innovation Association, FRUCT 2021. pp. 214–221. doi: 10.23919/FRUCT52173.2021.9435452.

[45] KVET, M. (2021). *Relational data index consolidation*. 28th Conference of Open Innovation Association, FRUCT 2021. pp. 215-221. doi: 10.23919/FRUCT50888.2021.9347614.

[46] KVET, M. (2022). *Study of duplicate tuple management*. pp. 3081–3088. doi: 10.1109/SMC52423.2021.9658726.

[47] KVET, M., KVET, M. (2021). *Relational pre-indexing layer supervised by the DB-index-consolidator background process*. Conference of Open Innovation Association, FRUCT 2021. doi: 10.23919/FRUCT50888.2021.9347573.

[48] KVET, M., MATIAŠKO, K. (2014). *Transaction Management in Temportal System.* IEEE Conference CISTI 2014 – pp. 868-873.

[49] KVET, M., MATIAŠKO, K. (2019). *Efficiency of the relational database tuple access*. INFORMATICS 2019 - IEEE 15th International Scientific Conference on Informatics, Proceedings. pp. 231–236. doi: 10.1109/INFORMATICS47936.2019.9119325.

[50] KVET, M., MATIAŠKO, K. (2020). *Analysis of current trends in relational database indexing*. International Conference on Smart Systems and Technologies (SST). pp. 109-114. doi: 10.1109/SST49455.2020.9264034.

[51] KVET, M., MATIAŠKO, K. (2021). *Data Loading and Migration Methods in the Cloud Environment*. In: 2021 Communication and Information Technologies (KIT). pp. 1–6.

[52] LI, S., QIN, Z., SONG, H. (2016). *A Temporal-Spatial Method for Group Detection, Locating and Tracking.* IEEE Access, volume 4.

[53] LIEBEHERR, J., OMIECINSKI, E. R., AKYILDIZ, I. F. (1993). *The effect of index partitioning schemes on the performance of distributed query processing*. In IEEE Transactions on Knowledge and Data Engineering, vol. 5, no. 3, pp. 510-522. doi: 10.1109/69.224201.

[54] LO, Y.-L., TAN, C.-Y. (2012). *A Study on Multi-Attribute Database Indexing on Cloud System*. Lecture Notes in Engineering and Computer Science. 2195, 299–304.

[55] LORENZINI, M., KIM, W., AJOUDANI, A. (2022). *An Online Multi-Index Approach to Human Ergonomics Assessment in the Workplace*. IEEE Transactions on Human-Machine Systems. doi: 10.1109/THMS.2021.3133807.

[56] MATIAŠKO, K., KVET, M. (2020). *Temporálne databázy – 1. vyd.* Žilinská univerzita, Žilina. ISBN 978-80-554-1662-5.

[57] MATIAŠKO, K., VAJSOVÁ, M., AND KVET M. (2017) *Pokročilé databázové systémy 1. diel.* EDIS.

[58] MEI, Y., JI, K., WANG, F. (2013). *A Survey on Bitmap Index Technologies for Large-Scale Data Retrieval.* 6th International Conference on Intelligent Networks and Intelligent Systems (ICINIS), Shenyang. pp. 316-319, doi: 10.1109/ICINIS.2013.88.

[59] PÖSS, M., POTAPOV, D. (2003). *Data compression in Oracle.* In: Proceedings 2003 VLDB Conference. Morgan Kaufmann, p. 937-947.

[60] PUSHPA, S., VINOD, P. (2007) *Binary Search Tree Balancing Methods: A Critical Study.* International Journal of Computer Science and Network Security 7. pp. 237-243.

[61] RADAIDEH, M.A. (2015). *A Distributed and Parallel Model for High-Performance Indexing of Database Content.* 26, 257–262 (2015). doi: 10.1080/1206212X.2004.11441747.

[62] RAOUF, A.E.A., ABO-ALIAN, A., BADR, N.L. (2021). *A Predictive Multi-Tenant Database Migration and Replication in the Cloud Environment*. IEEE Access. 9, pp. 152015–152031. doi: 10.1109/ACCESS.2021.3126582.

[63] ROLIK, O. et all. (2022). *Increase Efficiency of Relational Databases Using Instruments of Second Normal Form*. pp. 221–225. doi: 10.1109/ATIT54053.2021.9678605.

[64] SAYOOD, K. (2017). *Introduction to data compression*. Morgan Kaufmann.

[65] SHANBHAG, A. (2016). *An adaptive partitioning scheme for ad-hoc and time-varying database analytics*. Massachusetts Institute of Technology.

[66] SHARMA, V. (2020). *How indexing helps in improving performance of databases*. Retrieved from https://www.clariontech.com/blog/how-indexing-helps-in-improving-performance-of-databases.

[67] SPÄRCK JONES, K. (1974). *Automatic Indexing*. Journal of Documentation, Vol. 30 No. 4, pp. 393-432.

[68] TANG, Y., CHEN, L., LIU, J., LI, D. (2016). *Speeding up virtualized transaction logging with vtrans*. Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS. 0, pp. 916–923. doi: 10.1109/ICPADS.2016.0123.

[69] THAT, D.-H.T., GHAREHDAGHI, M., RASIN, A., MALIK, T. (2022). *LDI: Learned Distribution Index for Column Stores*. pp. 376–387. doi: 10.1109/BIGDATA52589.2021.9671318.

[70] TIMS, J., GUPTA, R., SOFFA, M. (1998). *Data Flow Analysis Driven Dynamic Data Partitioning*. In: Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers. pp. 75–90. Springer-Verlag, Berlin, Heidelberg.

[71] TUZHILIN, A. (2016). *Using Temporal Logic and Datalog to Query Databases Evolving in Time.* Forgotten Books.

[72] UNNIKRISHNAN, K., PRAMOD, K. (2009). *On Implementing Temporal Coalescing in Temporal Databases Implemented on Top of Relational Database Systems*. In: Proceedings of the International Conference on Advances in Computing, Communication and Control. pp. 153–156. doi: 10.1145/1523103.1523135.

[73] VINAYAKUMAR, R., SOMAN, K., MENON, P. (2018). *DB-Learn: Studying Relational Algebra Concepts by Snapping Blocks*. 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT). pp. 1-6, doi: 10.1109/ICCCNT.2018.8494181.

[74] WANG, M., XIAO, M., PENG, S., LIU, G. (2017). *A hybrid index for temporal big data*. Future Generation Computer Systems, 72, 264–272. doi: 10.1016/j.future.2016.08.002.

[75] WANG, Q., DU, Z., AND LIU, N. (2012). *Design and realization of database online migration*. 2nd International Conference on Computer Science and Network Technology, Changchun. pp. 1195-1198, doi: 10.1109/ICCSNT.2012.6526138.

[76] WANG, R., SALZBERG, B., LOMET, D. (2009). *Transaction support for log-based middleware server recovery*. Proceedings - International Conference on Data Engineering. pp. 353–356. doi: 10.1109/ICDE.2009.45.

[77] WESTMANN, T., KOSSMANN, D., HELMER, S., MOERKOTTE G. (2000). *The implementation and performance of compressed databases*. ACM Sigmod Record. 29(3). pp. 55-67.

[78] WU, E., MADDEN, S. (2011). *Partitioning techniques for fine-grained indexing*. IEEE 27th International Conference on Data Engineering. pp. 1127-1138. doi: 10.1109/ICDE.2011.5767830.

[79] ZYGIARIS, S. (2018). *Database Management Systems: A Business-Oriented Approach Using ORACLE, MySQL and MS Access*. Emerald Group Publishing.