PHYSICAL DESIGN OF MYSQL DATABASE

Ivan ŠUŠTER
Darjan KARABAŠEVIĆ
Dragiša STANUJKIĆ
Tamara RANISAVLJEVIĆ
Miodrag BRZAKOVIĆ

Abstract: *A database's physical architecture plays a critical role in guaranteeing effective data access and system performance. This research highlights the significance of workload analysis and index selection and prediction in improving query processing speed and database performance. It explores a number of indexing techniques, outlining the uses and advantages of B-Trees, Full-Text, Spatial, and Hash indexes, among others. In order to maximize database speed, it also covers denormalization and partitioning techniques including vertical and horizontal partitioning. By putting these strategies to cautious use and keeping an eye on things constantly, databases can be optimized and scalable to match changing system requirements.*
Keywords: *MySQL, Physical design, Indexing, Denormalization, Partitioning*

INTRODUCTION

In the world of databases, the physical design of the database plays a key role in ensuring the efficiency and speed of data access. Optimal functioning of the database requires a deep understanding of the context and proper application of its physical aspects. The process of physical design of the database begins after the development of conceptual and external schemes. Due to the dynamic nature of business requirements and technological advances, physical design is not a static process, but requires constant review and adaptation, ensuring that the efficiency and performance of the database is constantly improving in line with the evolving needs of the systems that the database supports.

This paper highlights the importance of predicting and selecting the right indexes, which are crucial for speeding up query processing and are the basis of physical database design. An equally important component of that process is database load analysis, as it is the load that enables effective targeting and optimization by seeing and understanding which queries the

database system uses most often, as well as how often those queries are executed.

In addition to indexing, there are other methods and techniques critical to optimizing database performance. Horizontal and vertical table partitioning are strategies that allow a database to manage large amounts of data more efficiently by dividing it into smaller, more manageable segments. Denormalization, although at first glance it may seem like a step back from the norms of database design, in certain situations it can significantly improve the performance of reading data. Those strategies and methods are necessary to achieve optimal performance, and are presented in more detail in this paper.

WORKLOAD

Workload in the context of databases refers to the total volume of queries, transactions and operations processed by the database system in a certain period of time (Almeida et al, 2023; Domaschka et al, 2023). This includes all types of database access and interaction, such as reading, writing, updating, and deleting data, initiated by users or applications. Workload definition enables database administrators and performance engineers to analyze and understand how database resources are being used, which is critical to optimizing system performance, capacity, and reliability. Workload analysis helps identify the most demanding operations and potential performance bottlenecks, which is necessary for effective capacity planning, scaling and application of appropriate optimizations (Koopmann, 2009; What is workload management).

The description and analysis of the workload includes the following (Ramakrishnan, Gehrke, 2002):

• A list of queries and how often they are used against all queries and updates,

• List of updates and their frequency,

• Target performance for each type of query or update.

For each inquiry it is necessary to specify:

• Which relations are accessed,

• What attributes are required in the SELECT statement,

• Which attributes are joined or selected in the WHERE statement and how selective those conditions are.

Each update must specify:

• The type of update (INSER, DELETE, UPDATE) and the relation

being updated,

• For the UPDATE command, the field to be updated.

The description of the query is very important in order to be able to carry out an physical design of the database, i.e. it is very important to decide when indexes will be used and which queries can gain speed during execution, which means that even when executing some other queries it can lead to an increase runtime which may be the case with update commands.

INDEXES

Indexes are data structures that allow efficient and fast access to data in the database. The main reasons for using the index are (Gao et al, 2023):

• Better performance: indexes enable the presenter to quickly access data.

• Uniqueness of data: indexes ensure that the values of the columns that are indexed are not repeated. So in databases, an index is usually automatically created for all primary keys.

During the physical design of the database, the choice of indexes is very important, that is, it is important to decide which indexes to create and on which columns. Performance depends on the indexes used as well as on which fields and/or relations they are used. It is also necessary to determine whether the index should be clustered, sparse or dense.

Another definition of indexes would be that they are a data structure that speeds up the search of tables in the database, i.e. it is not necessary to go through every row of the table every time you access the database. In addition to index selection, denormalization of the database schema should be considered, or which of the normal forms should be used for database design.

The MySQL database management system uses different types of indexes that provide query optimization and support different types of applications. The types of indexes used in the MySQL database management system are (Maesaroh et al, 2022):

• B Trees (eng. B-Tree) index - are usually used to facilitate efficient data retrieval and query processing. It is used with the =, >, >=, <, <=, BETWEEN and LIKE operators. It can be used in MyISAM, MEMORY and InnoDB storage engine.

• Full-Text Index - MySQL supports a full-text index which is necessary to enable the efficiency of text-based search operations. A full-text index is useful in applications that require advanced text search capabilities, that is, that involve large text data sets. It is used by InnoDB and MyISAM

storage engine.

• Spatial Index - these indexes are necessary for applications that include geographic or spatial data. Spatial indexes enable efficient processing of spatial queries, such as proximity searches and spatial joins, making them crucial for applications that use geographic objects. InnoDB and MyISAM use B-tree indexes for spatial data, while other storage engines use R-tree indexes.

• Hash index - MySQL also supports hash indexes, which are most effective for searching for exact data, ie when using the equal sign or <=> WHERE part of the query, because they provide fast access to data records. They are used by the MEMORY storage engine. All these types of indexes listed in the MySQL database management system meet different query requirements and data characteristics, enable efficient query processing and retrieval of data in the database.

In addition to the previously mentioned index types, it is also possible to create composite indexes (multi-column index), which consist of several columns (up to 16). With these indexes, the order of specifying the columns when creating the index is very important to ensure that the index will be used.

This index will be used only if the indexed columns used in the WHERE part of the query are listed in order from left to right, without skipping columns. So if a composite index is created over 3 columns, col1, col2, col3, the index will be used if the query is executed over col1, col1 and col2, col1, col2 and col3. When choosing an index type for a MySQL database, it is crucial to carefully evaluate the use case and the queries that the application frequently executes.

Different types of multi-column indexes, such as the ones above, provide different advantages based on the queries being processed, for example if you need to run multi-column queries frequently and know that the first column in the composite index will always be included. By understanding the unique application requirements, the most appropriate index type can be selected to improve query processing efficiency. By testing and monitoring different types of indexes and their impact on performance, the indexes that best fit the database can be identified.

CHOOSING THE RIGHT INDEXES

When creating an index, it is necessary to pay attention to the queries that need to be executed. So, for example, if in the WHERE part, selection

is made by some range, such queries gain the most performance if a B+ tree is used. If in the WHERE part the extraction is done using the equals sign, then the greatest benefit is obtained by using the Hash index.

Before adding indexes, it is also necessary to pay attention to the influence of them. When updating a column that has an index, then it is necessary to update the indexes as well. Also, if an INSERT or DELETE is performed, the indexes need to be updated if the value of the attribute changes.

The following instructions will explain how to select the index (Almeida et al, 2023; Maesaroh et al, 2022):

1. You should not create an index if it will not contribute to speeding up the execution of a query, including those that are part of updates. Whenever possible, choose indexes that will speed up more than one query.

2. Attributes mentioned in the WHERE clause are candidates for indexing.

• Extraction by exact value suggests that we should consider an index over the selected attributes, namely a hash index.

• Extraction by value range suggests that a B+ tree should be considered over the extracted attributes.

3. A multi-attribute search key should be considered in the following situations:

• A WHERE clause contains a condition on more than one attribute.

• If they enable index-only strategies (where access to relations can be avoided) in important queries. This can lead to attributes that would be part of the key even though they are not mentioned in the WHERE clause. When creating an index over search keys with multiple attributes, if interval queries are also accepted, one should be careful about the order of attributes in the search key.

4. At most one index over a given relation can be clustered, which greatly increases performance. So it is necessary to choose the clustered index carefully.

• Queries that contain a range can gain a lot from a clustered index. If several interval queries are used over the relation, which include different sets of attributes, when choosing which index should be clustered, the selectivity of the queries and their relative frequency in the workload should be considered.

• If the index enables an index-only strategy for the query that needs to be accelerated, then it does not need to be clustered. Clustering has an effect only when retrieving data.

5. The B+ tree is generally recommended because it is good for both

interval and exact value queries. A hash index is better in the following situations:

• A hash index should be used with a nested loop join, an indexed relation of inner relations, where the search key includes the columns on which the join is performed. The reason for this is that equality extraction is performed for each n-tuple in the outer loop.

• When there are very important equality queries and there are no interval queries, including attributes that are the search key.

6. After compiling the list of desired indexes, their impact on updates in the workload should be considered.

• If index maintenance slows down the update operation, consider dropping it.

• Keep in mind that adding an index can speed up data updates as well.

PHYSICAL DESIGN OF DATABASE

The need for the physical design of the database arises after the logical design of the database in order to optimize the operation of the database. The physical design of the database includes making decisions about how the data will be stored in the database as well as the choice of methods that will allow access to that data.

After the initial physical design of the database, it is necessary to continue monitoring the performance of the database during access, that is, data manipulation. Monitoring and tracking database data operations provides insight into which queries need to be optimized to further improve database performance. If necessary during operation, it is possible to make new settings such as adding new indexes or some of the partitioning techniques to improve performance. That is why it is very important to monitor and analyze the statistics and performance of queries against the data in the database. When physically designing a database, several aspects should be carefully considered to ensure an efficient design. These considerations are necessary to optimize performance, manage workloads, and adapt to changing demands.

During physical design, attention should be paid to the following key aspects (Teorey et al, 1986; Bahry et al, 2022; Rao et al, 2002):

1. Flexibility and future requirements - database design should represent the current state but also be flexible in order to adapt to future requirements.

2. Conceptual, logical and physical design - the physical design phase

includes conceptual design, logical and physical design, each of which plays a key role in shaping the structure of the database.

3. Query performance and workload - Physical design significantly affects query performance.

4. Continuous monitoring and adjustment - Continuous monitoring and improvement of the physical design and use of indexes is necessary in order to optimize the performance of the database.

DENORMALIZATION

In database architecture, denormalization is the addition of redundancy to the database schema with the goal of improving query performance and simplifying the retrieval of required data. This methodology involves deviating from the rules of database normalization in order to gain performance when accessing data that is important (Lee, Zheng, 2015).

Denormalization belongs to the physical design of the database at the logical level, where the normalized conceptual model of the database is taken as the starting point during denormalization. The fully normalized model is then denormalized, whereby normal forms are violated, data redundancy is introduced, and data integrity is violated (Buxton et al, 2009).

It is important to note that by introducing redundancy and violating data integrity, it is necessary to use the trigger mechanism to compensate for the consequences of denormalization, that is, to preserve data integrity. Below are the types of denormalization as well as examples of what this would look like and how to create triggers to preserve data integrity. The two types of denormalization that exist are denormalization of two entities in a one-to-one relationship and denormalization of two entities in a one-to-many relationship (Buxton et al, 2009).

If we take into account that there are entities employees and insurance policies, by denormalization those two entities can be merged into one in order to avoid frequent merging of tables in queries.

employee(emp_id, name, date_of_birth, address)
insurance_ policy(emp_id, policy_id, amount).
The functional dependencies of these entities are:
employee: emp_id -> name, date_of_birth
insurance_policy: emp_id -> policy_id, amount.

Denormalization of these tables represents an approach where it is necessary to combine the attributes of these two tables into one new combined table, joining them based on the primary key. This process as already

stated aims to optimize query performance and simplify data access by removing the need to use table joins in queries. Creating a new combined table, all information is easily accessible within a single table, thus increasing the efficiency of the database.

employee(emp_id, name, birth_date, address, policy_id, policy_amount).

Since it is a one-to-one connection, the integrity of the data remains the same and the new employee table remains in 3NF.

In the case of a one-to-many relationship, it is also possible to perform denormalization by joining tables and thereby eliminate the need to use joins when executing queries, the greatest efficiency is achieved if we perform denormalization on tables where joins and queries are often performed on those tables.

The following example shows a one-to-many relationship type and an example of denormalization.

employee(emp_id, name, address, *department_id*)
department(department_id, name)

The functional dependencies of these two tables are:

employee: emp_id -> name, address
department: department_id -> name

Denormalization will be performed by merging the employee and department tables into one called employee whose example is given below, and will also contain department data. In this way, the need to use joins when executing queries is successfully eliminated.

employee(emp_id, name, address, department_id, department_name)

If such a table were to be used, there would be possible problems when deleting a row in the table because data on the number and name of the department would be lost.

Denormalization, especially of the one-to-many type that involves joining tables to improve queries, should be approached with caution and used for specific cases, i.e. when query execution time is unsatisfactory and query time needs to be reduced. Implementing denormalization to optimize query performance should be carefully considered, although it can speed

up data retrieval, it leads to potential complexities in maintaining data integrity. In situations where query execution time is acceptable, performing denormalization is not recommended. Problems associated with denormalization such as increased storage requirements and data redundancy may outweigh the benefits of denormalization.

Denormalization can also affect data management, that is, changing, adding and deleting data. Therefore, it is necessary to carefully evaluate the advantages and disadvantages, as well as consider some other available physical database design strategies such as indexing. Only in cases where other approaches are inadequate, it is possible to consider denormalization as a potential solution and with caution about the impact on data integrity.

PARTITIONING

Table partitioning is a technique used to optimize database performance by dividing a single table into two or more smaller tables. Partitioning aims to reduce the amount of data that the database management system needs to process when executing queries. By dividing tables into several smaller tables, the system can use resources more efficiently, resulting in faster query processing time (Kumar, 2016).

There are two methods of partitioning:
- Vertical partitioning i
- Horizontal partitioning.

Both methods offer different advantages and are suitable for solving different problems. Both vertical and horizontal partitioning offer benefits in terms of performance and query optimization. The choice between them depends on factors such as the nature of the data, the query characteristics, and the scalability requirements of the database system. By carefully evaluating these factors, partitioning can be implemented to improve system performance.

Vertical table partitioning involves dividing the columns of the observed table into two or more tables, each containing columns based on their frequency of use in queries. This process optimizes data retrieval by separating frequently accessed columns into one table and less frequently accessed columns into another table. The division can continue even further, i.e. to more tables, with columns grouped according to the queries used. In each table resulting from vertical partitioning, the primary key of

the original table is included.

This primary key serves as a link that connects rows in partitioned tables. Using a primary key in each partitioned table makes it easier to manipulate the data in the partitioned tables. This approach improves database performance by reducing the amount of data that needs to be processed during query execution. Queries can be such that they search data in only one partition that contains the relevant columns. In addition, vertical partitioning can lead to better data storage efficiency and resource utilization, especially in cases where certain columns are larger or less frequently used or accessed than other columns.

Example table before partitioning:

employee (emp_id, name, address, date_of_birth, policy_id, policy_amount, policy_valid_from, policy_valid_to)

Appearance after partitioning:

employeesInsurancePolicy (emp_id, policy_id, amount, policies, policy_valid_from, policy_valid_to) *employeeData* (emp_id, name, address, date_of_birth).

After partitioning the tables, it is necessary to define a view that combines the data from the partitioned tables and thus creates a table that is identical to the initial table. Creating a view hides vertical partitioning and the need to join the resulting tables to get the initial table. However, in order to ensure the integrity and consistency of the data, stored procedures (or if available in the DBMS, triggers) for updating, adding and deleting rows from the view that will reflect changes in each partitioned table should be defined on the view created in this way. The MySQL database management system does not support vertical partitioning, that is, dividing a table according to columns into several smaller tables, so the example above is the only way to currently perform vertical partitioning in MySQL.

Horizontal partitioning, also known as row-based partitioning, is a database optimization technique that involves dividing a table into smaller partitions based on certain criteria. This method is especially useful for tables with a large number of rows, as it helps improve query performance.

When applying horizontal partitioning, the table is divided into different partitions, each of which contains a subset of rows that satisfy some conditions. One common partitioning criterion is time-based, where data

is partitioned based on date, one partition can contain data for each month, making separate partitions for January, February, etc. This temporal partitioning enables efficient data management, as queries can be made for specific time ranges without searching the entire table.

Data can also be partitioned based on the range of values within a particular column. For example, in the table of invoices, partitioning can be done based on the amount of invoices. Each partition would contain data for a certain range of invoice amounts. Using horizontal partitioning, data can be efficiently divided into multiple partitions, thus improving database performance and scalability.

HORIZONTAL PARTITIONING IN MYSQL DATABASE

In MySQL, in order to perform horizontal partitioning, it is necessary to use storage engines that support horizontal partitioning. What is important to note is that in the current version of MySQL (version 8.0) all partitions of a partitioned table must use the same storage engine.

In the current version of MySQL, the only storage engines that support partitioning are InnoDB and NDB (NDB cluster). With the fact that partitioning by key or linear key is possible with the NDB storage engine, while other types are not supported for tables using this engine (Overview of Partitioning in MySQL).

When partitioning a table, the default storage engine is used. However, in order to use another engine, it is necessary to explicitly state which one is used in the part for creating the table, before any partitioning options. This ensures that the specified engine is applied to the entire table, this is important since it is impossible to partition the table if the storage engines do not match.

When managing data within a partitioned table, handling old or unnecessary data is relatively simple. It is common to remove a specific partition such as a partition that contains outdated data. This method makes it possible to efficiently remove unnecessary data leaving the rest of the table accessible.

The ability to store data that meets certain criteria on one or more partitions brings significant benefits, especially in optimizing search operations. When searching with a condition in the WHERE clause, the search is automatically limited to the relevant partition and thus eliminates the need to scan unnecessary data.

Table partitioning also allows you to customize the partitioning

scheme as needed. This means that data can be reorganized to better meet the criteria of frequently used queries, even if the original partitioning was not designed to optimize certain queries.

The types of partitioning in MySQL are the following:

1. RANGE - table partitioning based on a range, where each partition includes rows whose values fall within a defined interval. These intervals should represent continuity, meaning that each range is adjusted to the next range so that there is no overlapping of range boundaries. To define these limits, the VALUES LESS THAN operator is used, which enables precise definition of the upper limit for each partition. Later, if there is a frequent increase in the range, ALTER TABLE can be used to add new partitions with a new range of values. Another example that is very useful in practice is partitioning by date, that is, it is possible to partition the table so that rows related to a specific time period are displayed, which is shown in the following example (the same table as in the previous examples is used).

2. LIST - similar to RANGE partitioning, because it is necessary to explicitly define the value based on which the partitioning is performed. However, the difference is in how the range of partitioning is defined, with RANGE partitions are determined by the adjacent range of values, while with LIST partitioning, the membership of the partition is determined based on the membership of the column values within predefined lists of values. This type of partitioning allows more flexibility since it is not necessary to have a continuous range of values. For this type of partitioning, the clause PARITION BY LIST(col) is used, where col is the column on the basis of which the membership of a partition is determined and which returns an integer, also after each partition comes the definition of the list of values of each partition using VALUES IN (*list_item1, list_item2, ...*), where list list of comma-separated integers. When defining the list, it is necessary to take into account all possible values that can be entered in the column by which the partitioning is performed, in the support column there is an error that there is no partition for that value.

3. COLUMNS - the type of partitioning allows the use of multiple columns on the basis of which partitioning is performed, there are also two variants of this partitioning, namely RANGE COLUMNS and LIST COLUMNS partitioning.

a. RANGE COLUMNS is very similar to RANGE partitioning but allows the definition of partitions based on multiple columns. This functionality provides the possibility of partitioning data based on more complex criteria that include several attributes. Another advantage of this way of

partitioning is that it provides the possibility of using other types of data in addition to integers and dates, namely strings. This capability provides a partitioning solution that is adaptable to the specific needs and characteristics of the data set.

b. LIST COLUMNS partitioning also builds on LIST partitioning and, like RANGE COLUMNS partitioning, allows multiple columns to be used to create partitions. In addition, it allows the use of other data types such as date and datetime, as well as strings.

4. HASH - partitioning is used for even distribution of data over a predetermined set of partitions. While RANGE or LIST partitioning requires a partition specification for each column value or set of values, HASH partitioning simplifies this process by automatically determining the appropriate partition. This type makes it easy to use by requiring only the specification of the partitioning or hashing column and the desired number of partitions. Partitioning is done on the basis of the column, i.e. the value given by the module, which represents the number of given partitions. A general formula would be if *bp* represents the number of partitions given, *i* the value of the expression and *n* the number of the partition where the expression will be placed then .

5. KEY - key partitioning in MySQL works similarly to hash partitioning, with a difference in the way the partitioning is performed. While hash partitioning relies on a user-defined hashing expression and KEY partitioning uses the hash function provided by MySQL, also instead of the HASH keyword, KEY is used when defining partitioning. KEY partitioning specifies a list with zero or more column names. It is important to note that columns marked as a partition key must either be part of or contain the entire primary key of the table if it exists. If no column is specified for the partition key, MySQL uses the primary key of the table. Another significant difference from other partitioning methods is that the columns used for key partitioning are not limited to integer values only. This flexibility allows a wider range of data types to be used for partitioning.

6. SUBPARTITION is a further division of partitions. Further partitioning is possible only by range or list and it is possible to further use hash or key partitioning. Each of the partitions must have the same number of subpartitions, it is also possible to name the sub-partitions, and if one subpartition is named, the other subpartitions must also be named.

CONCLUSION

Physical design of a MySQL database is a key process in ensuring its optimal performance, scalability and reliability. Through careful consideration of factors such as indexing, denormalization, partitioning, and caching mechanisms, it is possible to fine-tune the database structure to meet specific operational requirements. The essence of this process lies in the regular monitoring and analysis of database performance, in order to identify opportunities for improvement and implement the necessary changes.

In addition to direct technical adjustments, it is important to consider the possibilities of upgrading the hardware, which can significantly contribute to the performance of the base. The success of physical design is not only in the application of individual techniques, but also in strategic planning and continuous monitoring of the database, to ensure that the changes lead to the desired improvements.

Therefore, we conclude that continuous evaluation of database performance, using appropriate techniques and tools for monitoring and analysis is the foundation of the physical design process. Such an approach enables not only the identification of areas for improvement, but also ensures that the database is constantly evolving to meet both the current and future demands of the systems it serves.

REFERENCES

Almeida, D., Lopes, M., Saraiva, L., Abbasi, M., Martins, P., Silva, J., & Váz, P. (2023, August). Performance Comparison of Redis, Memcached, MySQL, and PostgreSQL: A Study on Key-Value and Relational Databases. In 2023 Second International Conference On Smart Technologies For Smart Nation (SmartTechCon) (pp. 902-907). IEEE.

Bahry, F. D. S., Amran, N., Putri, T. E., & Ramli, M. I. (2022). *Database design of the malaysia public figures web archive repository: a social and cultural heritage web collections*. Collection and Curation, 41(4), 133-140.

Buxton S., et al. *Database Design: Know it all*, Morgan Kaufmann, 2009.

Domaschka, J., Volpert, S., Maier, K., Eisenhart, G., & Seybold, D. (2023, April). Using eBPF for Database Workload Tracing: An Explorative Study. In Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (pp. 311-317).

Gao, P., Chen, Q., Xie, X., & Wang, C. (2023, May). Research on Performance Optimization of MySQL Database. In 2023 IEEE 3rd International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA) (Vol. 3, pp. 869-872). IEEE.

Koopmann, J. 2009. "What Is Your Definition of Databse Workload?" in Database Journal.

Available: https://www.databasejournal.com/oracle/what-is-your-definition-of-database-workload/ (Accessed: 14.01.2024.)

Kumar, A. S. (2016, August). Performance analysis of MySQL partition, hive partition-bucketing and Apache Pig. In 2016 1st India International Conference on Information Processing (IICIP) (pp. 1-6). IEEE.

Lee, C. H., & Zheng, Y. L. (2015, October). SQL-to-NoSQL schema denormalization and migration: a study on content management systems. In 2015 IEEE International Conference on Systems, Man, and Cybernetics (pp. 2022-2026). IEEE.

Maesaroh, S., Gunawan, H., Lestari, A., Tsaurie, M. S. A., & Fauji, M. (2022). Query optimization in mysql database using index. International Journal of Cyber and IT Service Management, 2(2), 104-110.

"Overview of Partitioning in MySQL" in MySQL Documentation, Available: https://dev.mysql.com/doc/refman/8.0/en/partitioning-overview.html (Accessed: 20.01.2024.)

Ramakrishnan, R., & Gehrke, J. (2002). *Database management systems.* McGraw-Hill, Inc.

Rao, J., Zhang, C., Megiddo, N., & Lohman, G. M. (2002). Automating physical database design in a parallel database. Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data.

Teorey, T. J., Yang, D., & Fry, J. P. (1986). *A logical design methodology for relational databases using the extended entity-relationship model*. ACM Computing Surveys, 18(2), 197-222.

"What is workload management?" in Microsoft Learn. Available: https://learn.microsoft.com/en-us/azure/synapse-analytics/sql-data-warehouse/sql-data-warehouse-workload-management (Accessed: 14.01.2024.)

Notes on the authors:

Ivan ŠUŠTER, B.Sc., is a M.Sc. student at the Faculty of electronic engineering, University of Niš. E-mail: ivansu995@gmail.com

Darjan KARABAŠEVIĆ, Ph.D. is a Full Professor and Dean of the Faculty of Applied Management, Economics and Finance, University Business Academy in Novi Sad. E-mail: darjan.karabasevic@mef.edu.rs

Dragiša STANUJKIĆ, Ph.D. is a Full Professor of Information Technology at the Technical Faculty in Bor, University of Belgrade. E-mail. dstanujkic@tfbor.bg.ac.rs

Tamara RANISAVLJEVIĆ, B.Sc. is a M.Sc. student at the Faculty of Applied Management, Economics and Finance, University Business Academy in Novi Sad. E-mail. tamara.ranisavljevic@gmail.com

Miodrag BRZAKOVIĆ, Ph.D., is a Full Professor and Council President at the Faculty of Applied Management, Economics and Finance, University Business Academy in Novi Sad. E-mail. miodrag.brzakovic@mef.edu.rs