The Journal of Systems & Software

Extending Hybrid SQL/NoSQL Database by Introducing Statement Optimisation Component

--Manuscript Draft--

Manuscript Number:	JSSOFTWARE-D-23-00202
Article Type:	VSI:SEAA-2022 [MGE - Regina Hebig]
Keywords:	hybrid database; optimisation; SQL; NoSQL; Oracle; MongoDB
Abstract:	The business of contemporary organisations often includes different subdomains, for which it is neither easy nor optimal to decide on using an exclusive database type. The hybrid SQL/NoSQL databases encompass various types of databases unified into a single logical database. At the same time, they provide use benefits of working with the SQL and the NoSQL databases simultaneously. In recent years, there has been an increase in research that deals with the challenge of hybrid databases' query optimisation. This trend opened up possibilities for analysing the optimisation of other data manipulation statements (INSERT, UPDATE and DELETE). For this paper, a process model for the application of hybrid statements' optimisation was designed. Also, this paper presents the extended architecture for work with the hybrid SQL/NoSQL database. The architecture extension contains the newly developed statement optimisation component (SOC). The article gives an overview of the current optimisation rules stored in the Optimisation Rules repository. The set of rules is easily extensible. The use cases chosen for testing were executed over the hybrid with the SOC component and compared to the non-optimised statements executed without the SOC component. Conducted tests indicate particular decreases in the average execution times of the optimised statements.

Extending Hybrid SQL/NoSQL Database by Introducing Statement Optimisation Component

- Highlights -

Hybrid databases use different database types (SQL and NoSQL) as their components

We developed a process model for applying optimisation techniques to the statements

A newly created Statement Optimisation Component extended a hybrid SQL/NoSQL database

Tests show the decreased execution times on a hybrid with a new SOC component

Title page with author details

Extending Hybrid SQL/NoSQL Database by Introducing Statement Optimisation Component

Srdja Bjeladinovic (corresponding author)
University of Belgrade, Faculty of Organizational Sciences, Department of Information Systems,
Address: Jove Ilica 154, 11000 Belgrade, Serbia
E-mail: srdja.bjeladinovic@fon.bg.ac.rs

Extending Hybrid SQL/NoSQL Database by Introducing Statement Optimisation Component

Srdja Bjeladinovic (corresponding author)

University of Belgrade, Faculty of Organizational Sciences, Department of Information Systems, Address: Jove Ilica 154, 11000 Belgrade, Serbia

E-mail: srdja.bjeladinovic@fon.bg.ac.rs

Abstract:

The business of contemporary organisations often includes different subdomains, for which it is neither easy nor optimal to decide on using an exclusive database type. The hybrid SQL/NoSQL databases encompass various types of databases unified into a single logical database. At the same time, they provide use benefits of working with the SQL and the NoSQL databases simultaneously. In recent years, there has been an increase in research that deals with the challenge of hybrid databases' query optimisation. This trend opened up possibilities for analysing the optimisation of other data manipulation statements (INSERT, UPDATE and DELETE). For this paper, a process model for the application of hybrid statements' optimisation was designed. Also, this paper presents the extended architecture for work with the hybrid SQL/NoSQL database. The architecture extension contains the newly developed statement optimisation component (SOC). The article gives an overview of the current optimisation rules stored in the Optimisation Rules repository. The set of rules is easily extensible. The use cases chosen for testing were executed over the hybrid with the SOC component and compared to the non-optimised statements executed without the SOC component. Conducted tests indicate particular decreases in the average execution times of the optimised statements.

Keywords:

hybrid database, optimisation, SQL, NoSQL, Oracle, MongoDB

1. Introduction

Hybrid databases have been developed to satisfy the growing need of contemporary businesses for storage, access and processing of data, regardless of the source and degree of data structuredness. In recent years, the hybrid database term has been increasingly used in literature. Different authors describe the meaning of the term in different ways. Section 2 discusses the hybrid database term. However, regardless of the various interpretations of the term hybrid databases, the common idea behind database hybridisation is to integrate databases of different types into a single logical database. This approach in developing data-intensive systems enables the use of suitable Database Management System (DBMS) representatives for each business domain, or more precisely, for each business subdomain. Contemporary organisations' business often unifies business functions that use strict and in advance defined data

structures with other functions that use highly flexible data structures. Apart from the simultaneous use of data with different levels of structuredness, various aspects may have higher or lower importance, depending on the business needs. For example, we can observe an organisation's information system, which entails monitoring and execution of financial transactions (with the banks, business partners, employees etc.) and marketing promotions on social media. We use different potential criteria for the successful execution measurement of each function. Regarding financial transactions, it is necessary to provide data integrity. In the case of social media promotion, data integrity is often secondary-significant to the data availability and speedy analytical processing.

Before the emergence of hybrid databases, the challenge of using data with different levels of structuredness had one of the following outcomes: limiting organisations to using an exclusive database type for all subsystems or using different types of databases for each of the subsystems (or for the group of similar subsystems). In the previous example, as in many other examples of business, it is not always easy nor optimal to choose one data management technology. The choice of exclusive database type gives the organisation benefit of using it for suitable subsystems. At the same time, other subsystems have the limitations of using a non-suitable or non-optimal database type.

The second mentioned approach to the problem, the usage of different types of databases whose operation is not fluidly connected into a single logical database, implies additional expenses of connections, integration and unified administration. Compared to the two prior approaches, hybrid databases enable integration and concurrent use of different technologies. Hybrid database users benefit from using the best functionalities of different database types. They do not have to worry about their integration since different types of databases represent the components of the single logical (hybrid) database. Hybrid database usage enables the engagement of suitable data storage and management technology for every business subdomain. The analysed papers from Section 2 emphasise this. The most common and comprehensive representatives of the hybrids are the SQL/NoSQL hybrid databases.

This paper represents the natural continuation of the research into the hybrid databases' design approach (Bjeladinovic, 2018) and its architecture improvement presented in the article (Bjeladinovic et al., 2020). The aim of this paper is the hybrid database extension and improvement by introducing the optimisation techniques for its statements. The following research questions led to the hybrid SQL/NoSQL database extension presented in this paper:

1) Which statements are feasible to optimise, and with which techniques in order to improve the hybrid SQL/NoSQL database? How to define the application process of the hybrid statement optimisation techniques?
2) How to adjust the architecture suitable for work with the hybrid database, with the aim of enabling integration and uniform usage of the Statement Optimisation Component (SOC) with other components?
3) How do the implemented optimisation techniques influence the performance, more precisely the duration of the statement execution, based on the example of the Oracle/MongoDB hybrid database?

In order to present the existing research in the field and the original results obtained in the process of answering the research questions, the paper has the following organisation. Section 2 overviews the existing directions of hybrid databases' design and use. Section 3 deals with the first research question by presenting the process model designed for applying optimisation techniques of this approach. Section 4

describes the architecture extension, with a newly created component for statement optimisation. Description of the SOC component's role in the architecture and explanation of its functioning aim to answer the second research question. Section 5 lists the use cases chosen for testing and comparing the average statements' execution times for the hybrid SQL/NoSQL databases without the dedicated optimisation component and for the hybrid SQL/NoSQL databases with the newly created SOC. The Oracle/MongoDB hybrid database was used as the test environment. Section 6 contains experimental results. Section 7 consists of conclusions and gives directions for future research.

2. Related work

In general, hybrid databases can be defined as integrated data systems comprised of multiple autonomous databases (Vyawahare et al., 2018) or as databases that integrate different types of databases into a single logical database (Bjeladinovic, 2018). By reviewing their presence on the market (SolidIT, 2023a), it can be concluded that the dominant databases are still relational (also commonly called SQL databases after the standardised query language they use). However, NoSQL databases, suitable for working with large amounts of data, are increasingly being used (Sudhakar, 2018). Today, big companies such as Facebook, Amazon and Google use SQL and NoSQL databases to complement each other (Faraj et al., 2014). The hybrid SQL/NoSQL databases unify the use benefits of the SQL and NoSQL databases for the purposes they are designed for by overcoming individual limitations typical for particular database types (Zhang et al., 2022).

At times in literature, the NoSQL databases are referred to as non-relational (Goyal et al., 2015; Gyorodi et al., 2015; James and Asagba, 2017). However, non-relational databases can be generally treated as a broader term, including other types of databases (Jatana et al., 2012). The increased popularity of the NoSQL databases directly affected the focus of databases' hybridisation in recent years. The focus is shifted towards integrating the two mentioned types of databases, that is, towards the design and use of hybrid SQL/NoSQL databases (Villari et al., 2016). Therefore, the authors (Goyal et al., 2015) state that hybrid databases aim to use data from relational and non-relational databases and to provide conjoint results in a single output. The paper (Martinez-Mosquera et al., 2020) defines the term hybrid database or just the hybrid as 'systems where there are several databases implemented that can be relational and/or NoSQL.'

The review in the field identified a few groups of papers of interest. The first group is comprised of the articles that contribute towards setting hybrid databases' general postulate (i.e. design, integration, uniform use). This group of papers set the foundations for hybrid databases. The articles whose research focus is on different aspects of performance measurements and database optimisation of different types represent the second group of research papers. In addition, the significance of these papers is reflected in the fact that different database types can be components of the hybrid. Finally, the final group of manuscripts directly discusses the hybrid databases' optimisation challenge.

The authors (Roy-Hubara and Sturm, 2020; Kalayda, 2021; Bender et al., 2022) performed a trend review of the contemporary databases' designs, including common, current and future development directions. The hybrid design and use challenges originate from the heterogeneous characteristics of different database types integrated into a single logical entity. The authors (Bjeladinovic, 2018; Zečević et al., 2018) dealt with developing dedicated design methodologies for hybrid databases. The authors (Aleem et al., 2019) analysed software test techniques for the hybrid databases. The authors (Villari et al., 2016)

highlighted the use benefits of the hybrid SQL/NoSQL databases and presented dedicated components of the extended ER model. These components are useful while modelling hybrid databases. Primarily theoretical integration possibilities of the relational (MySQL) and graph database (Neo4j) are discussed in the papers (Vyawahare et al., 2018; Vyawahare et al., 2019). In their paper (de la Vega et al., 2018), the authors focused on the diversity of NoSQL models (key-value, document-oriented, column family, graph) and highlighted that relational databases' traditional design approaches cannot be directly applied. They presented the Mortadelo framework based on the model-driven transformation process. It transforms the general data model into the intermediate logical model (specific for some of the four NoSQL models). The latter should then be transformed into the implementation code of the particular DBMS. The authors (Sokolova et al., 2019) presented the migration approach from the SQL into the hybrid SQL/NoSQL database using ontology to define data schema. In their paper, the authors (Abdelhedi et al., 2017) presented how a conceptual model could describe Big Data stored in the NoSQL database. Unlike the Mortadelo framework that uses specific logical models for each NoSQL database type, the authors (Abdelhedi et al., 2017) created a generic logical layer suitable for work with three types of NoSQL databases (document-oriented, column family, and graph). By applying the QVT rules, this layer enables efficient transformation execution into the physical model, with the decreased influence of the destination NoSQL database's technical specifications. The authors (Zečević et al., 2018) analysed the metamodel integration possibilities of different database types. Through the application of the lightweight extension approach, this paper describes the way of adding new elements and constraints to the existing metamodels. By applying one conceptual, one logical and one physical model, this approach enables the integration of different types of databases. The authors depicted this in the example of the system comprising one relational database and two document-orientated databases. Some authors (Mershad and Hamieh, 2021) approached the topic of hybrid databases from the aspect of domain-specific language (DSL). The limitation of the domain-specific languages is the lower prevalence than of the standardised SQL, supported by leading manufacturers or relational databases.

The authors (Čerešňák and Kvet, 2019; Fraczek and Plechawska-Wojcik, 2017) compared query execution in relational and non-relational databases. The paper (Fraczek and Plechawska-Wojcik, 2017) compared the three DBMS's login and usage performances (PostgreSQL, MongoDB and Cassandra), while the article (Čerešňák and Kvet, 2019) analysed in detail query execution of the three most common SQL database representatives (Oracle, MySQL and MS SQL Server) and four representatives of the NoSQL databases (MongoDB, Redis, Cassandra and GraphQL). The mentioned authors obtained more or less expected results between the SQL and NoSQL database types under the requirements set before testing. The gap decrease between the SQL and NoSQL databases' performances the authors (Liu et al., 2016) have achieved by introducing a dedicated binary format for JSON. The applicability of this solution is reflected in the hybrid databases whose components support JSON format. However, this also results in usage limitations on the hybrids that contain NoSQL databases without JSON support. The authors (Kemper and Neumann, 2011) approached the optimisation of different databases in use from the aspect of the gap elimination that emerges while using traditional OLTP and OLAP systems. They suggested the creation of a hybrid OLTP & OLAP system that would contain data versioning. As stated by the authors, introducing data versioning would enable the separation of data manipulation from query execution while using a hybrid system's database for both purposes. This solution allows for the execution of the BI queries "*on an arbitrarily current database snapshot system*" while eliminating the consumption of the resources derived from the additional activities necessary for data adjustment and transfer from the traditional OLTP systems into the OLAP systems (Kemper and Neumann, 2011). From the theoretical aspect, the authors (Thiry et al., 2018) dealt with the optimisation of a large amount of data with a different degree of structuredness. In addition to the presented mathematical model, the authors showed postulates of the DSL, which is based on the unification concept with the aim of easier information search.

The author (Scheuermann, 2010) researched the hybrid database design directions. These were inspired by finding a solution to the challenge of hardware components' optimisation and their specificity use with the aim of offloading database operators. The author has been using parsing and rewriting components of the existing DBMS (the paper mentions PostgreSQL as a potential candidate) and optimiser (whose main part is cost optimiser that calculates operations' expense based on the data from the dictionary). With those components, the author dealt with the execution plan optimisation considering different execution engine types. The paper focuses on the execution plan adjustment to the hardware components without considering hybrid SQL/NoSQL database optimisation specificities. The authors (Owaida et al., 2017) dealt with the optimisation of the hybrid databases' hardware components, with a focus on CPU/FPGA, while the priority on CPU/GPU was given in the papers by the authors (Breß et al., 2013; Cremer et al., 2017; Gowanlock et al., 2019). Even though, in general, the mentioned papers discuss the optimisation of hybrid systems and hybrid databases, they will not be further analysed in this paper, given their focus on the hardware aspect.

The authors (Pang et al., 2021) dealt with hybrid optimisation made of 'classical' databases and MapReduce. They analysed the advantages and limitations of databases and the MapReduce systems and highlighted the benefits of using hybrids made up of the mentioned types of repositories. Finally, they presented a dedicated and improved version of the query optimiser named AquaPlus. The purpose of the presented optimiser is to use database features as much as possible (like index and partitioning), with the aim of reducing the amount of data needed to be processed by the MapReduce hybrid component. The authors (Khan et al., 2019) had a compatible approach. In their paper, they researched the effect of physical optimisation, predominantly partitioning, on the average time of query execution in the SQL database (Oracle DBMS). After that, the authors compared the SQL database's improved performance with the graph database (Neo4j). They concluded that the gap was decreased primarily in the complex queries (subqueries and JOINs). A noticeable step towards improving query optimisation in a system made of heterogenous databases of different types was made by the authors (Sellami and Defude, 2018). Even though they do not explicitly use the term 'hybrid databases' but 'virtual data source' instead, they have focused their research on the hybrid domain, whose components are relational and NoSQL databases. The authors suggested the introduction of the mediation component, whose aim is to optimise queries for efficient execution over multiple databases of different types. They used a joint schema to describe all data sources in the system. In addition, they enabled parallel execution over data sources and optimal plan generation by using dynamic programming. However, these authors focused solely on queries, and they did not deal with other statements. The authors (Li and Gu, 2018) developed a useful solution to the nested query optimisation problem over the hybrid operation. In the mentioned paper, they were solving the integration problem of MySQL, MongoDB, and Redis as the hybrid database components and simultaneous query execution over the relational and NoSQL databases. They presented the Multiple Sources Integration architecture (MSI) that supports databases' integration. Also, they graphically presented the communication between components for optimisation with the SQL parser on one and the SQL router on the other side. In addition to explaining algorithm logic that was used for the nested query optimisation, the authors state that "*the expression of the query conditions must be carried out according to the distributive law, which also needs to be simplified based on the Espresso algorithm… and it is planned to be discussed in another paper.*" (Li and Gu, 2018). Even though it is stated that the presented architecture supports optimisation, the article focuses on one optimisation segment, more precisely on the nested queries. This paper, as well as others analysed in this Section, created a solid base for query optimisation.

The papers presented in this Section confirmed the purposefulness of further hybrid database optimisation research. It opened up possibilities for and motivated authors to expand the optimisation

research of not only queries but also other statements for a hybrid database's data manipulation while adjusting the hybrid architecture for this purpose.

3. Process model for statement optimisation of hybrid SQL/NoSQL database

Given that the authors of the presented papers took into consideration queries only, additional opportunities emerged for the research of optimisation possibilities of other data manipulation statements (INSERT, UPDATE and DELETE). Research in this paper encompasses queries and statements for insertion, update and deletion, which complete the most commonly executed CRUD (CREATE, READ, UPDATE, DELETE) operations over data. These are, based on the SQL standard terminology, classified into the DML (Data Manipulation Language) statements.

All leading manufacturers of relational database management systems (RDBMS) support the standardised SQL language. SQL is a declarative query language which enables uniform usage of data manipulation statements, creation and update of different types of objects of a relational scheme, as well as control of transaction execution. The procedural extensions of SQL language are not standardised. The manufacturers use specific extensions (i.e. Oracle use PL/SQL, and Microsoft use T-SQL) instead. Unlike the SQL databases, the NoSQL databases do not have a standardised query language, not even for CRUD operations. The complexity of the challenge is that in addition to the lack of unified language for all NoSQL databases, there is no agreement about the language of particular NoSQL databases' subtypes (key-value, document-oriented, column family, graph databases). Therefore, none of the NoSQL database subtypes have their unique language. The absence of the NoSQL database standardised language made it difficult to optimise statements, which are executed in the NoSQL components of the hybrid SQL/NoSQL databases. Also, the lack of NoSQL language standardisation limited the scope of possible solutions. Further language unification by the NoSQL databases' manufacturers will open possibilities for additional extension and improvement of the solution suggested by this paper.

With the aim of introducing optimisation support for all four aforementioned statements, a dedicated process model was designed. Figure 1 depicts the UML activity diagram for the statement optimisation process. The designed activity diagram starts with the statement input, followed by its decomposition and analysis.

Some of the hybrid database's main strengths are a single logical data model usage (common for all hybrid components), and user disburden of knowing what component stores needed data (Bjeladinovic et al., 2020). Given the scenario in which the input statement is executed over multiple target components, i.e. different DBMSs of the hybrid SQL/NoSQL database, the next activity is statement decomposition on its integral parts. The approach presented in this paper, as well as the mentioned approach it extends, uses the standardised SQL language for the statement input over the hybrid database exclusively. SQL language, popular among users, and the supported architecture eliminate the need to know or use other domain-specific languages for each particular hybrid component. The user has access to all the data as if they are stored in a single logical relational database. The user exclusively uses the SQL syntax, regardless of whether the destination of the input statement is the SQL component, NoSQL component or both. A detailed description of the architecture with the newly developed component for the statement optimisation, which enables the stated operation, is given in Section 4.

The optimisation preparation activities start after analysing and decomposing the statement and its integral parts. Depending on the input statement (SELECT, INSERT, UPDATE, DELETE), the process execution transfers to the appropriate branch following the decision node in the activity diagram. The

decision node is used for exclusive branching depending on the input statement type. Exclusive branching is present since the execution of different types of nested statements (i.e. UPDATE with subquery or DELETE with subquery) is not supported by the current version of the architecture. These are the limitations of the current version, as well as the direction for further.

The optimisation recommendations for all hybrid DBMSs that use SQL language are given within one activity of a single branch, marked with [SQL DBMS]. It is the uniformity of SQL language that made this possible. Each statement type will have a single [SQL DBMS] branch.

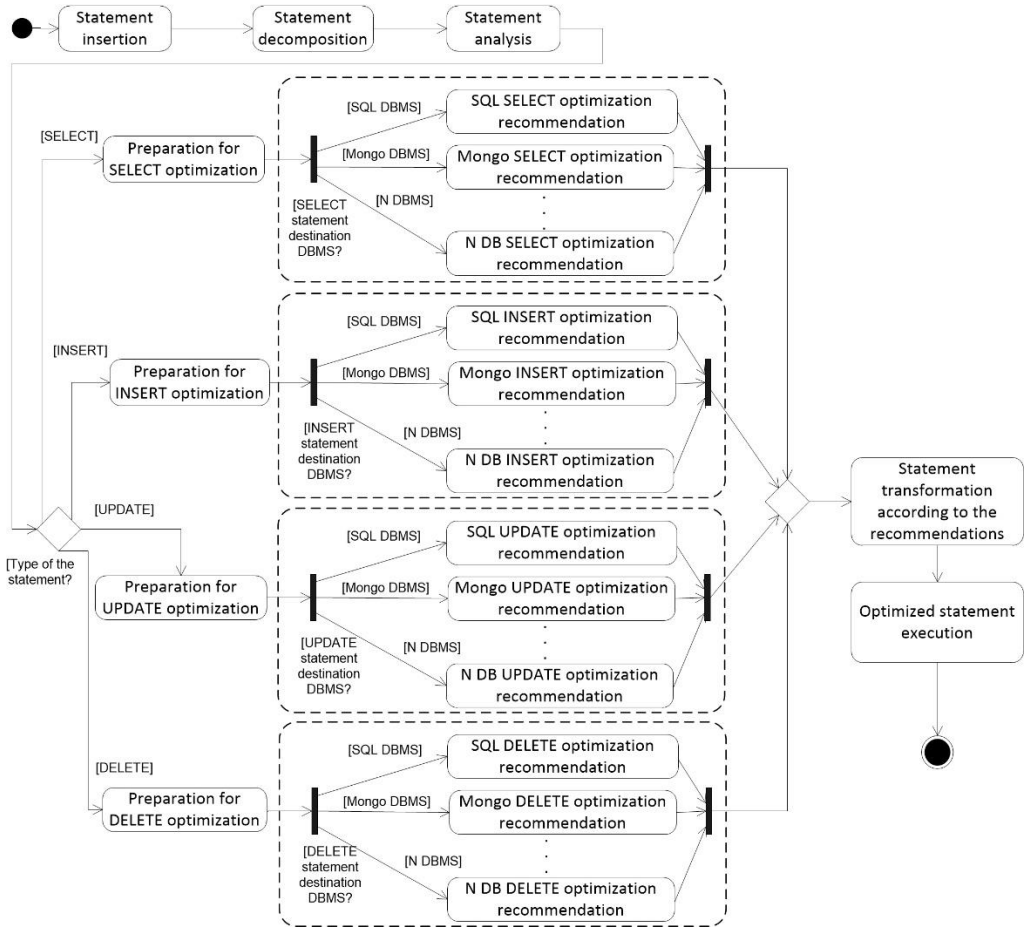


Figure 1 – The UML activity diagram for applying optimisation recommendations on the SQL and NoSQL components of the hybrid SQL/NoSQL database

A different situation is observed with the NoSQL components. Given the absence of a standardised NoSQL language, it is necessary to give specific recommendations for each exact NoSQL DBMS, which is depicted in Figure 1 through the appropriate flows with conditions ([Mongo DBMS]…[N DBMS]). Each of these flows gives specific recommendations for a particular NoSQL DBMS in the syntax it supports. A 'three dot' symbol on the diagram suggests that the hybrid database can consist of an arbitrary number of different NoSQL components, and N DBMS represents the Nth NoSQL DBMS. The fork node enables parallel optimisation of different parts of the input statement. These parts could be executed into various destination components of the hybrid database (i.e. SELECT statement that displays data from the SQL component and one or more NoSQL components).

Branch flows with accepted recommendations meet in the join nodes for each statement type. Next, all flows from all statements meet in the merge node. After merging flows, statement transformation takes place. It follows the identified recommendations for optimisation. If necessary, the keywords of the entered SQL statement are mapped according to the language of the destination database, i.e. hybrid component. Syntax transformation does not take place if the destination database is SQL, given that the input statement is already in SQL language. Thus prepared and (according to the accepted recommendations) optimised statement is then executed. Statement execution represents the last activity on the diagram depicted in Figure 1.

The presented diagram displays the general logic of introducing the statement optimisation techniques over different components of a hybrid database. A hybrid Oracle/MongoDB database is the chosen representative to depict some of the supported statement optimisation techniques enabled by this approach. As the name suggests, this hybrid consists of two components: Oracle DBMS, the representative of the relational DBMS, and MongoDB, the representative of the document-oriented DBMS. The selection was made based on the popularity rankings of these DBMSs. At the time of writing this paper, Oracle was the most popular SQL system (SolidIT, 2023b), while MongoDB was the most popular document-oriented and NoSQL overall system (SolidIT, 2023c). Use cases chosen for testing were executed over the hybrid. Section 6 consists of the average execution time for each tested use case. The supported recommendations for optimising certain statements, classified into Oracle/MongoDB hybrid database components, are given in Table 1. These recommendations represent supported statement transformation techniques in the current version of the system. The volume of the supported optimisation rules is extendible and will be expanded in future research.

The process model in Figure 1 is comprehensive, and it depicts optimisation techniques' application activities for all DML statements. Even the first version of the hybrid SQL/NoSQL database incorporated optimisation techniques for the SELECT statement (query writing syntax, indexes, partitioning etc.). This is already shown through the execution results of the tested queries in earlier papers (Bjeladinovic, 2018; Bjeladinovic et al., 2020). For this reason, the focus of this paper is on the optimisation of other DML statements (INSERT, UPDATE, DELETE) through the application of the newly developed SOC component, as well as on the effects of stated rules usage on chosen use cases.

For the optimisation of the INSERT statement, supported syntaxes are INSERT ALL (for the SQL component) and BULK INSERT (for the MongoDB component). The principle is the same, and the expected benefit is in the shortened average execution time of the optimised statement compared to the initial statement. This expectation is based on inputting multiple records in one DBMS call, with the syntax that eliminates multiple INSERT. The UPDATE statement optimisation rules contain a recommendation for parallel execution by using the PARALLEL hint for the SQL component. For the MongoDB component, using the dedicated `updatemany()` method instead of the multiple `updateone()` method should provide better results. Each input statement should satisfy preconditions for the optimisation rule to be applicable. A detailed description of these preconditions is in Section 4. Even though the DELETE statement supports parallel execution, as INSERT and UPDATE in SQL database, Table 1 shows a more efficient optimisation technique. However, its application scope is noticeably smaller. When the deletion of all table records is needed (the DELETE statement without the WHERE clause), the TRUNCATE statement can be executed while preserving the table itself, its structure and its constraints. The expected benefits of the average execution duration are reflected in the more efficient realisation of the DDL statement (TRUNCATE belongs to this category) instead of the multiple DELETE statement execution. The limitation of using TRUNCATE is that `rollback` is not accessible, given that `auto-commit` follows TRUNCATE by default (as well as other DDL statements). In the described case of data deletion, the `drop()` method can be used for MongoDB. Although this action implies collection deletion, during new

document input into the non-existent collection, the stated collection is automatically created and thus does not represent a significant resource cost for INSERT. The other option would be the use of the `remove()` statement.

Table 1 – Supported statement optimisation techniques for the hybrid Oracle/MongoDB database

Type of statement	Statement scenario	Recommendations for statement optimisation (SQL component)	Recommendations for statement optimisation (NoSQL component)
INSERT	Inserting multiple rows	INSERT ALL syntax. *The decrease in the number of calls to the database, compared with the execution of multiple but individual INSERT statements.*	BULK INSERT syntax. *The decrease in the number of calls to the database, compared with the multiple calls of the insertone() method.*
UPDATE	Updating multiple rows	PARALLEL update. *Under certain conditions, hint PARALLEL enables the optimisation via parallel execution of the UPDATE statements instead of the default sequential execution.*	updatemany(). *Update multiple documents that satisfy the filter specified as the first argument instead of executing individual updateone() methods numerous times.*
DELETE	Deleting all rows (DELETE without a WHERE clause)	TRUNCATE statement. *Using the mentioned DDL statement gives the benefits of quickly deleting a large amount of data while preserving the table structure.*	DROP method. *The DROP method enables the quick deletion of all data, as well as the collection. Due to its structure flexibility, the new record insertion automatically recreates the collection and thus does not represent a significant resource cost for INSERT.*

It is important to note that the supported statement optimisation techniques cannot always apply. The focus is on optimising statements whose execution affects "multiple" records (for insert, update or delete statements). Therefore, the term 'recommendation' is on the activity diagram. At the same time, the recommendation represents the crucial part of each optimisation rule, as shown in Table 2.

Whether the recommendation will be applied depends on the statement type and fulfilment of the specific preconditions. Despite all syntax preconditions fulfilling (the number of statements, the existence of particular clauses and similar), sometimes additional conditions must be met. For applying PARALLEL onto the UPDATE statement (it is the same for INSERT or DELETE), it is necessary to enable parallel execution of the DML statements on the system or session level by running the command `(alter system/session enable parallel dml)`. Preconditions such as this can affect the application outcome of the given recommendation (similar to how statistical data of statement execution and calculated cost of accessing the data in alternative ways affect the index application in the query). Therefore, specific rules come down to the recommendation, and it is impossible to give general enough and for the execution engine an utterly binding way of applying these recommendations. On the other

hand, the absence of PARALLEL hint usage due to the unsupported parallel statement execution does not affect the success of the statement realisation. As in the case of stating the hint for non-adequate and non-optimal indexes during query execution, the stated PARALLEL hint gets neglected, and the statement is executed without an error occurring due to the forwarded hint.

To a certain extent, decision-making is automated (for supported techniques) through the hybrid database dedicated architecture that supports the presented process model and new statement optimisation component. Regardless, statement optimisation is a complex process. It is often limited by adjustment options as well as the influence on the way the optimiser and execution engine of the DBMS work. The hybrid database extension with the newly created SOC component uses the optimisation advantages of particular DBMSs, with limitations within those DBMSs. Described condition is evident when individual DBMSs are observed, especially within hybrid databases. If the provided hint in the SQL database has a higher value of execution cost compared to its alternatives, the execution engine will not use the hint. The same principle follows a hybrid database because the stated derives from the execution engine of the specific component.

At this time, creating a fully comprehensive solution for optimising all types of databases that a single hybrid can encompass is extremely challenging to do. It is not an easy task to determine the number of domain-specific languages used by all NoSQL databases currently available on the market. Even if there were an estimate, without the language standardisation by the leading NoSQL manufacturers, there would not be a comprehensive solution for implementing the optimisation techniques for the whole hybrid database.

Therefore, this paper aims to demonstrate the feasibility of extending the dedicated architecture for work with a hybrid database by introducing the newly developed SOC components that operate in accordance with the designed process model in order to improve performance.

4. Extending hybrid SQL/NoSQL database with Statement Optimisation Component (SOC)

The architecture presented in the paper (Bjeladinovic et al., 2020) was taken as the starting point of the system that uses the hybrid database. This architecture enables integration and uniform use of all hybrid database components. It gives the user benefits of using different technologies, as well as the convenience of working with a single logical database. The limiting factor of the initial architecture, which also represents the direction for further research, is the lack of support for the statement optimisation. This paper presents the extension of the initial architecture with the design of the SOC component. Figure 2 shows the extended SQL/NoSQL database architecture with the SOC component.

Pictured architecture represents the extension of the traditional three-tier architecture. Without going into the details of user interface organisation and implementation (graphical interface, statement input console and similar), the purpose of its presentation layer is to display data and to enable statement input and execution by the user. The middle layer is represented through the Wrapper, and it contains earlier established components necessary for providing support with the hybrid database. SQL language was chosen for the entire hybrid database and all types of databases that constitute its components, regardless of whether a particular database type supports SQL language. The motivation was to provide the comfort of using a single query language and to free the user from thinking about which component has requested data.

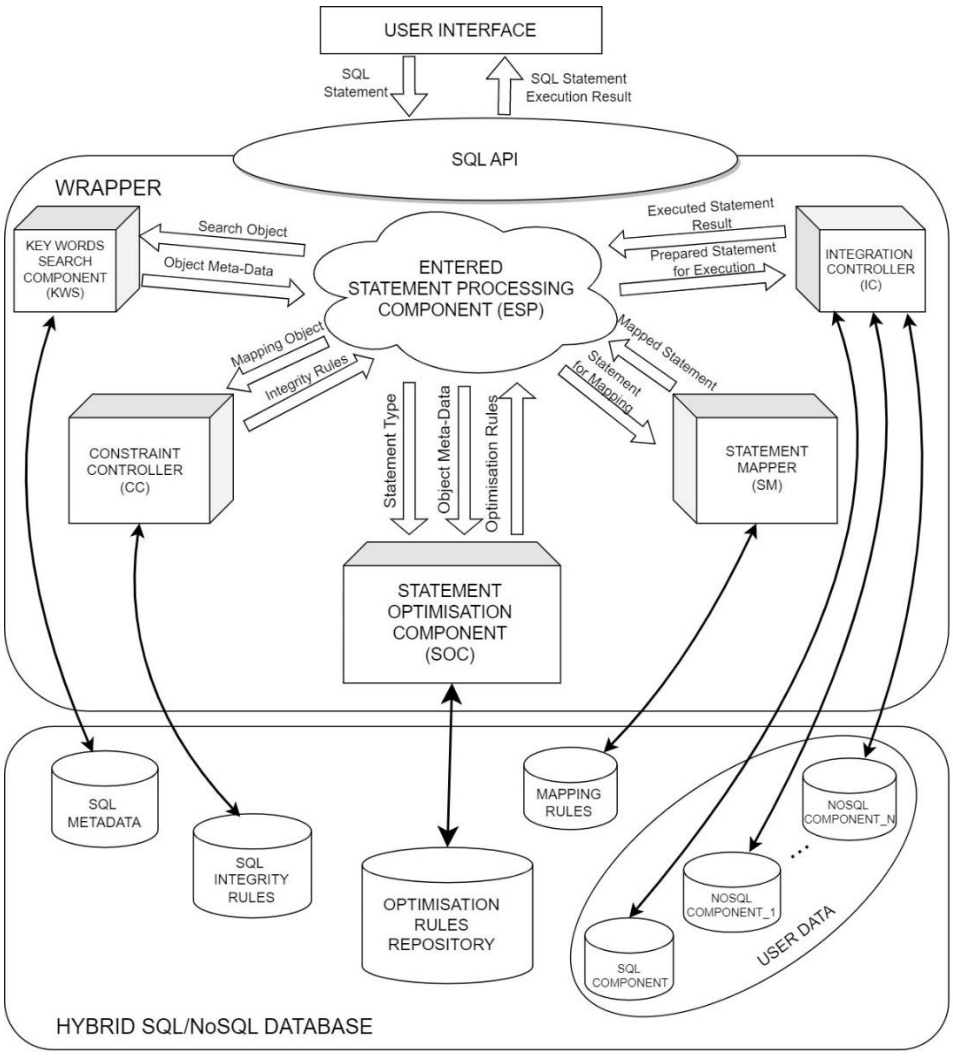


Figure 2 – The SQL/NoSQL database architecture extended with the newly developed component for the statement optimisation (SOC)

SQL API is in charge of communication to the presentation layer. The Wrapper manages all middle-layer components, including SQL API. Following the acceptance of the SQL statement, it is necessary to analyse, decompose, optionally map and forward the statement or its parts for execution to the hybrid destination components (i.e. specific DBMSs unified by the hybrid). This activity is in the jurisdiction of Entered Statement Processing Component-e (ESP), the central communication component of the Wrapper. In order to achieve this, EPS communicates with other Wrapper components, sends them requests, processes their return values and manages the whole process from the reception of the SQL statement to its execution and display of the returned results to the users. From the function description, in the most general sense, the EPS component is most similar to the traditional three-tier architecture controller. The components EPS communicates with are (present in the initial architecture) Key Words Search Component (KWS), Constraint Controller (CC), Statement Mapper (SM), and Integration Controller (IC), as well as, in this paper introduced, the newly developed Statement Optimisation Component (SOC).

After analysing the SQL statement entered by user, the EPS component communicates with the KWS component by sending the objects' names (from the FROM clause). It receives metadata about the objects as a response from the KWS component. The metadata contains the object type (table, column, key-value

pair etc.) and the database type the object belongs to (SQL, document-oriented NoSQL, column family NoSQL etc.). It also contains the specific DBMS in which the object is implemented (Oracle, MongoDB, Cassandra etc.). Here, it is necessary to highlight one of the essential characteristics of the hybrid database: the whole hybrid database, with all its components, represents a single logical database. Given that all hybrid objects, regardless of what component they belong to, are integrated with the joint data model of a single logical database, a hybrid can't contain two objects with the same name but of a different type. Therefore, for every object name forwarded to the KWS, the EPS receives a single object type, a single database type it belongs to and one specific destination DBMS. Based on the return values, the ESP component will decide if it is necessary to execute statement mapping. When entered statement or part of that statement has a destination in a database type that doesn't support SQL language, a mapping will occur. In contrast, when entered statement and its integral parts have a destination in a database that supports SQL language, a mapping doesn't happen.

The next component the ESP addresses in the communication chain is the CC. The CC is in charge of centralised management of all hybrid components' integrity rules. Since SQL databases provide a higher degree of constraint control (Nance et al., 2013), it is the SQL databases' integrity rules (that encompass entity integrity rules and referential integrity rules) that are implemented as common characteristics for the whole hybrid database. The lack of NoSQL support for the mentioned rules is overcome by the integrated placement of constraints in the dedicated hybrid's SQL component for storing integrity rules (named Integrity Rules). It contains the rules of entity integrity, i.e. it takes care of the primary key of each object (columns or fields, depending on the type of database) and its complexity (whether it contains one or more columns or fields). In addition to the entity rules, the Integrity Rules repository contains the referential integrity rules (i.e. foreign keys). It keeps data about referenced and referencing columns (or fields), regardless of which types of database objects participate in the referencing. This functionality overcomes the lack of integrity rules support in the NoSQL databases and enables fluid referencing between the SQL and NoSQL database objects, as well as between different types of NoSQL objects.

Metadata, Integrity Rules, Mapping Rules and the newly added Optimisation Rules represent dedicated SQL databases for metadata storage. These should not be confused with databases that are part of the hybrid database and contain user data. The reason for choosing the SQL databases for metadata storage is based on the strict ACID properties' support, imminent for the consistent use of metadata.

In the earlier version of the hybrid architecture, after obtaining the integrity rules from the CC component, the ESP component communicated with the SM component by sending the entered statement and metadata of its objects.

The extension of the improved architecture operating logic enables the communication of the ESP component with the newly developed SOC component. The earlier version didn't contain the SOC component. Instead, when needed and after receiving the integrity rules from the CC component, the ESP component sent the request to the SM component to perform statement mapping into the domain-specific language. In the extended architecture, the ESP component communicates with the newly developed SOC component before interacting with the SM component. The ESP component sends the statement type and metadata of destination objects to the SOC component. The SOC component accesses the newly introduced Optimisation Rules repository. The Optimisation Rules repository contains necessary preconditions, recommendations and application rules of specific optimisation techniques for particular statements.

Table 2 – Optimisation Rules repository structure

RULE_ID	STAT_TYPE	COMP_TYPE	MULTI_ROWS	WHERE_CLAUSE	...	RECOMMENDATION
101	INSERT	SQL	YES	N/A		INSERT_ALL
102	INSERT	NoSQL/Mongo	YES	N/A		BULK_INSERT
...						
201	UPDATE	SQL	YES	YES/NO		PARALLEL
202	UPDATE	NoSQL/Mongo	YES	YES/NO		UPDATE_MANY
...						
301	DELETE	SQL	YES	NO		TRUNCATE
302	DELETE	NoSQL/Mongo	YES	NO		DROP
...						

Table 2 shows the structure of the Optimisation Rules repository. This table contains selected examples of the optimisation rules for INSERT, UPDATE and DELETE statements of the hybrid Oracle/MongoDB database.

Selected examples do not show the optimisation rules for queries because earlier architecture versions have already supported them. The examples in this paper are focused on supporting other DML statements in order to present how the newly developed SOC component operates, as well as to show the effects of its use.

If the hybrid database introduces additional components, new rules, in the form of new records in the Optimisation Rules, are added for each statement type of each new database. The stated repository, in addition to the specific optimisation recommendation (RECOMMENDATION column), contains a statement type (STAT_TYPE column), a component type (COMP_TYPE column) and preconditions for the recommendations' use. Each precondition corresponds to a column of the same name. In Table 2, optimisation rules examples have depicted preconditions in two columns (MULTI_ROWS and WHERE_CLAUSE columns), and other rules can have additional preconditions (marked with '…'). Only essential columns for the chosen examples are shown in Table 2. Columns with preconditions represent a specific optimisation technique known at Oracle under the name Hard-coded values. In the Hard-coded values column, the CHECK constraint defines the range of valid values. It contains the value 'YES' for preconditions for which fulfilment is obligatory, while the value 'NO' is for preconditions in which fulfilment must not be satisfied. The value 'YES/NO' is for optional preconditions, i.e. that precondition doesn't influence the use of the particular rule, and the value 'N/A' is for preconditions not applicable to the observed rule.

For the specific INSERT ALL rule to be applied, it is necessary that the precondition of inserting multiple rows (column MULTI_ROWS has a 'YES' value) is fulfilled. In contrast, the prerequisite WHERE_CLAUSE does not apply to this rule (WHERE_CLAUSE has the value 'N/A') because the INSERT syntax does not support the WHERE clause. Similarly, the same precondition needs to be fulfilled (MULTI_ROWS) for the insert of multiple rows into the MongoDB component, while, once again, WHERE_CLAUSE is not applicable.

TRUNCATE and DROP are respective optimisation rules for the SQL and MongoDB DELETE statements for deleting all rows without filtering the records. That is why, for the observed DELETE rules, column MULTI_ROWS has a 'YES' value, and WHERE_CLAUSE has a 'NO' value.

The optimisation rules for the UPDATE statement of Oracle and MongoDB components of the observed Oracle/MongoDB hybrid are PARALLEL and UPDATE_MANY, respectively. Both rules are applicable for multiple rows updates (column MULTI_ROWS has a 'YES' value) and can be applied regardless of whether all records are updated or just filtered data (column WHERE_CLAUSE has a 'YES/NO' value).

Based on forwarded metadata, the SOC component determines if it is possible to perform statement optimisation and how. The SOC component reads applicable optimisation rules of the input statement from the Optimisation Rules repository and forwards them to the EPS component. After receiving the return values from the SOC component, the statement execution flow of the new hybrid with SOC is equivalent to the process in the earlier architecture version. The ESP component sends statements for mapping to the SM component. By reading the rules from the Mapping Rules repository, it maps statements into the domain-specific languages and returns them to the ESP. The ESP component sends mapped statements to the IC component. The IC component has the role of managing and controlling the execution of the statements in one or more components. The IC sends the statement execution results and feedback to the ESP component. The ESP component then adjusts the results format to be user-friendly and forwards it to the presentation layer, i.e. to the appropriate user interface. The described activities encompass the whole process from the SQL statement input, analysis, decomposition, optimisation, optional mapping, statement execution in the hybrid's destination component and user notification.

5. The use of the hybrid database with SOC on Oracle/MongoDB example

The data model, Figure 3, was developed to demonstrate the use of the current version of the hybrid SQL/NoSQL database with the new SOC component. The UML Class diagram depicts created model. The domain of the model is online search and product payment. Figure 3 represents only a part of the model, which was needed for the realisation of the use cases chosen for testing (for example ordering process was not necessary to show). The model consists following classes: User, User_status, Payments, Searches, Archived_payments, and Archived_searches, and it is implemented in two versions of the system: the previous hybrid Oracle/MongoDB database without SOC and the current hybrid Oracle/MongoDB database with SOC.

To unconditionally ensure the consistency of sensitive data, information about users, their statuses and payments are stored in the SQL component of the hybrid. For users' searches, availability and fast reporting have a higher level of importance than the necessary consistency, so the mentioned part of the system (Searches) is implemented in the NoSQL component of the hybrid system.

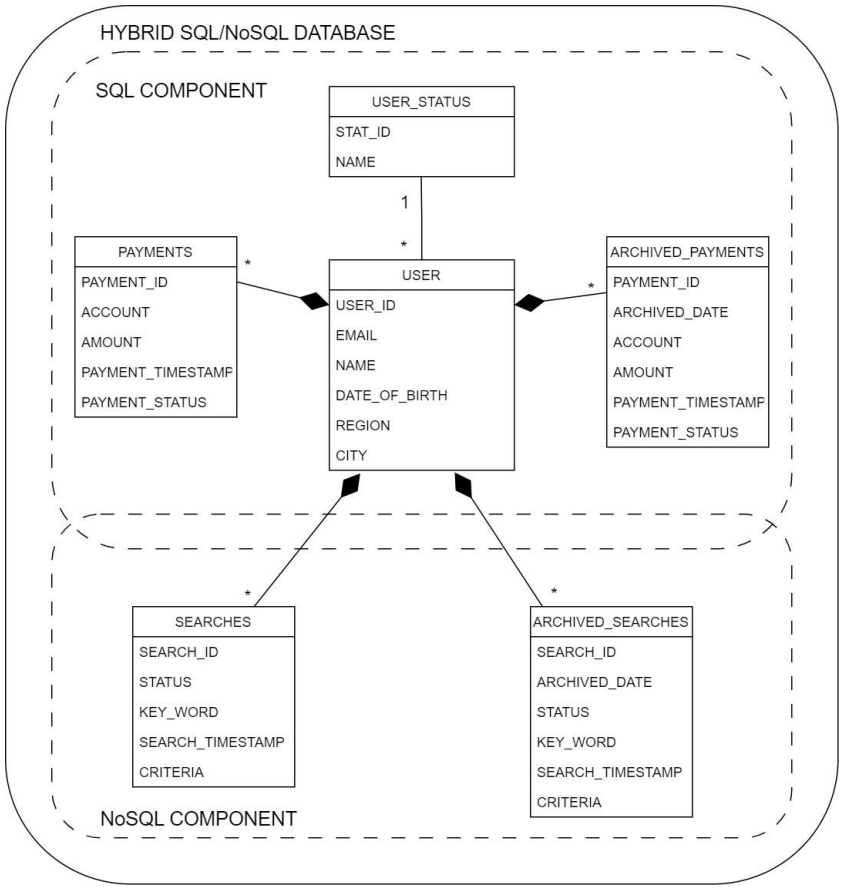


Figure 3 – UML Class diagram for tested domain

Payments and Searches have identification and existence dependence on the entity User. In the diagram, they make a possessive Composition relationship with the User class. Archived_payments is a table inside the SQL component, while Archived_searches represent documents in the NoSQL component of the hybrid database. A large amount of data is cyclical, in certain time intervals, being input into Archived_payments and Archived_searches from Payments and Searches, respectively. This is how two use cases are profiled, one for data insertion (usually several dozens of thousands of records) into the SQL component (table Archived_payments) and the other one for the entry of, once again, a large amount of data into the NoSQL component (Archived_searches structure). After the realisation of the mentioned use cases, records are deleted from Payments and Searches, which represent use cases for deleting all records from the SQL and NoSQL components, respectively. For the SQL and NoSQL components' update, chosen use cases were the user status change (foreign key) in the User table (the SQL component) as well as the performed searches update (the NoSQL component). Table 3 shows the snapshot of use cases chosen for testing based on the hybrid Oracle/MongoDB components. For every use case, Table 3 displays the use case id, a statement type, a destination component of the hybrid, a statement description and a statement syntax before and after applying the supported optimisation rules. The statement optimisation rules, shown in Table 2 and described in Section 4, were applied to these six chosen use cases.

Table 3 – Chosen use cases for testing

Use case id	Statement type and hybrid component	Statement description	Statement before optimisation	Statement after optimisation
UC_1	INSERT into SQL component	INSERT rows into archived_ payments	INSERT INTO archived_payments VALUES… … INSERT INTO archived_payments VALUES…	INSERT ALL INTO archived_payments… … INTO archived_payments…
UC_2	INSERT into NoSQL component	Insert rows into archived_ searches	db. archived_ searches.insertOne(…) … db. archived_ searches.insertOne(…)	var bulk = db.archived_ searches.initializeOrderedBulkOp(); bulk.insert(…); … bulk.insert(…); bulk.execute();
UC_3	UPDATE SQL component	Users with the status_id = 1 update to status_id = 2	UPDATE user SET status_id = 2 WHERE status_id = 1	UPDATE /*+ PARALLEL(4)*/ user SET status_id = 2 WHERE status_id = 1
UC_4	UPDATE NoSQL component	All searches with the status ‘Accepted’ update to values ‘Done’	db.searches.updateOne({Status: "Accepted" }, {$set: { Status: "Done"}}) … db.searches.updateOne({Status: "Accepted" }, {$set: { Status: "Done"}})	db.searches.updateMany({Status: "Accepted" }, {$set: { Status: "Done"}})
UC_5	DELETE from SQL component	Delete all rows from Payments	DELETE FROM payments	TRUNCATE TABLE payments
UC_6	DELETE from the NoSQL component	Delete all rows from Searches	db.searches.deleteMany({})	db.searches.drop()

The current architecture version of the hybrid SQL/NoSQL database is still a prototype, and not all functionalities are entirely implemented. However, the chosen use cases will demonstrate how it

operates. The six selected use cases were first executed over the previous version of the hybrid architecture without the SOC component and, after that, over the extended version of the hybrid architecture, which contains the SOC component.

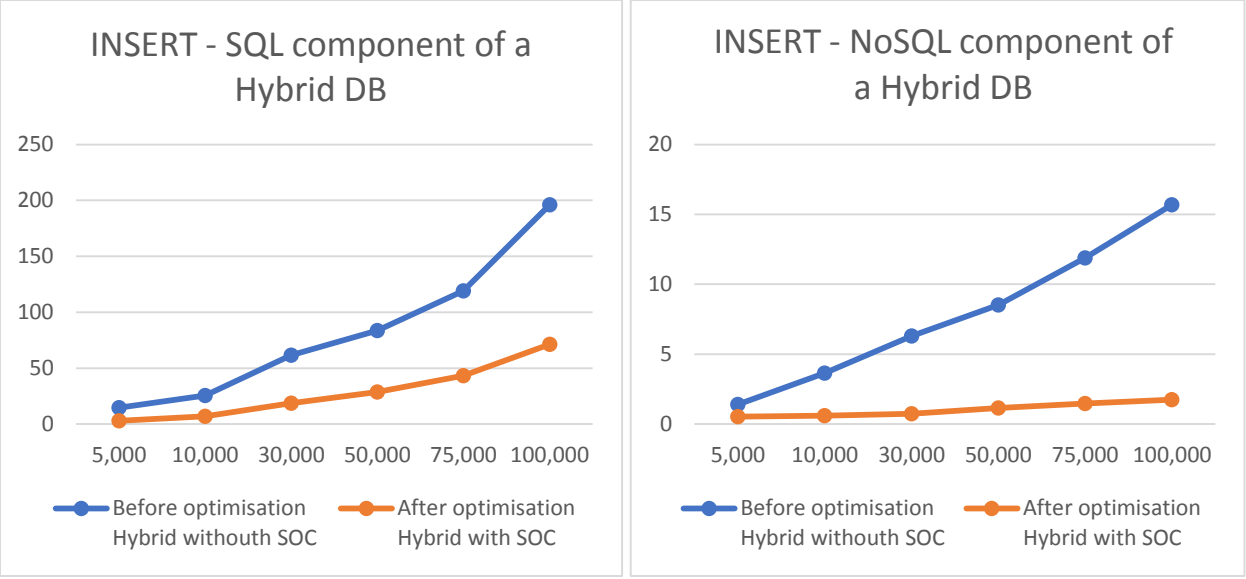
Tests were carried out on the PC with Intel i7 CPU, with a 2.9 GHz speed, with 16 GB RAM and SSD hard disk. The testing system has Windows OS, Oracle DBMS version 19c for the SQL component and MongoDB version 3.6 for the NoSQL component of the hybrid. The average statement execution time was taken as the performance indicator.

NetBeans IDE was used for statements inputs, executions and time measurements. Each test had 12 iterations. In order to eliminate the outliners, tests with the shortest and the longest execution times for each statement were discarded. The average time contains the execution times of the remaining ten iterations.

Measurements of use cases UC_1, UC_2, UC_5 and UC_6 were conducted on datasets of 5,000, 10,000, 30,000, 50,000, 75,000 and 100,000 records. The INSERT and DELETE use cases for SQL (UC_1 and UC_5), and NoSQL (UC_2 and UC_6) components were performed over the same amount of records. Use cases UC_3 and UC_4 were tested over 100,000, 300,000, 500,000, 750,000 and 1,000,000 records. Increasing the dataset relative to the remaining statements was chosen due to the nature of the described model. Since archived tables are periodically filled (INSERT), and operational tables are emptied (DELETE), a larger amount of records was selected for UPDATE to cover the amount of data that can be moved in several cycles. What follows is the display and the analysis of the average execution times of the use cases chosen for testing, focusing on the execution times before and after the optimisation rules use.

6. Experimental results

The average measured execution times of the use cases chosen for testing is shown in Figure 4, Figure 5 and Figure 6. The X-axis of the diagrams shows the number of records affected by the particular statement execution. The Y-axis shows the average statement execution time in seconds. In addition, every chart has two parts. The left part of the diagram, marked as (a), shows the duration of the statement execution in the SQL (Oracle) hybrid component before and after optimisation. The right part of the diagram, marked as (b), shows the average duration of the observed statement execution in the NoSQL (MongoDB) hybrid component before and after optimisation. Before-optimisation statement reflects the statement execution in a hybrid database without SOC (previous architecture), and the after-optimisation statement presents the statement execution in a hybrid database with SOC (extended architecture).



(a) *(b)*

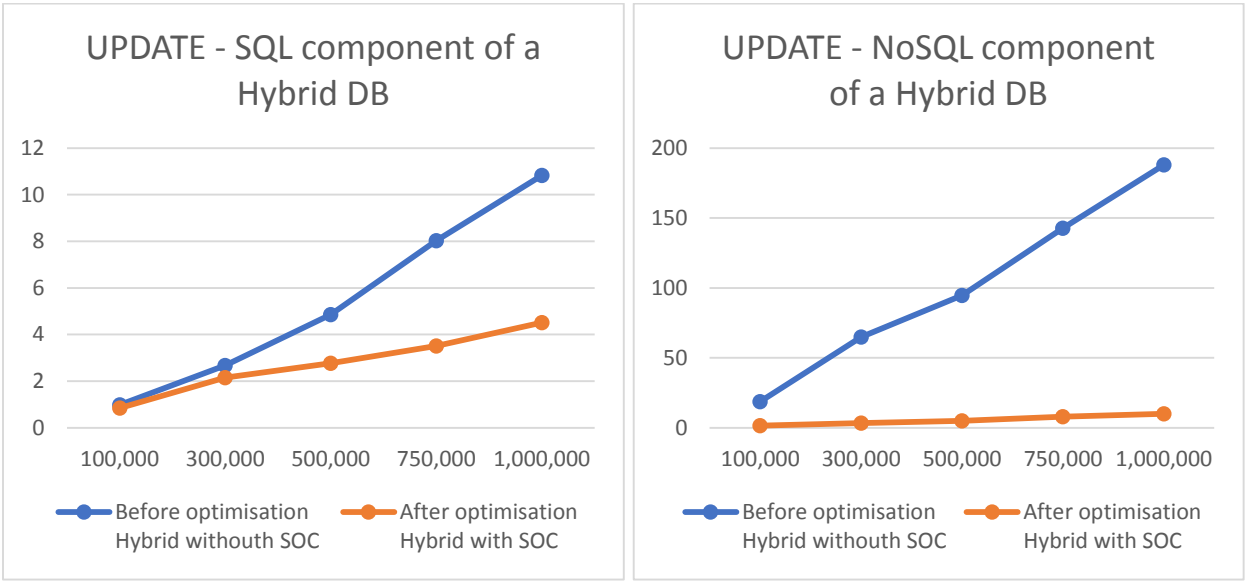
Figure 4 – The average measured execution times of use cases UC_1 (a) and UC_2 (b)

Figure 4 shows the average execution time of the INSERT statement. In the SQL (Oracle) component, the average execution times of the INSERT statement before optimisation were 14.836 (5,000 records), 25.543 (10,000 records), 61.547 (30,000 records), 83.721 (50,000 records), 119.236 (75,000 records) and 196.008 (100,000 records) seconds. Following the optimisation, INSERT in the Oracle component lasted on average 3.018 (5,000 records), 7.123 (10,000 records), 18.856 (30,000 records), 28.641 (50,000 records), 43.378 (75,000 records) and 71.23 (100,000 records) seconds. The average times for data insertion into the MongoDB component before optimisation were 1.414 (5,000 records), 3.655 (10,000 records), 6.295 (30,000 records), 8.527 (50,000 records), 11.885 (75,000 records) and 15.697 (100,000 records) seconds. However, after optimisation, it took 0.529 (5,000 records), 0.601 (10,000 records), 0.735 (30,000 records), 1.147 (50,000 records), 1.473 (75,000 records) and 1.739 (100,000 records) seconds.

The optimised INSERT statement uses INSERT ALL and BULK INSERT for the SQL and NoSQL component, respectively, and achieves shorter execution times over a hybrid without SOC, as expected. In addition, the average data insertion time in the NoSQL component is noticeably shorter than in the SQL component on a comparable number of records. The explanation is that records in the SQL components have a strict schema structure and additional constraints to satisfy.

It is important to point out that the INSERT statement with SQL optimisation recommendations has undergone a slight syntax adjustment. The INSERT ALL statement is generally created by concatenating the INTO table_name clause to the one INSERT statement. That way, multiple uses of the INSERT statement are eliminated. Although this approach has a fixed cost in concatenating the INTO clauses before executing the statement, the concatenation itself does not require much time. However, the limitations of this technique are the significant increase in the number of statement characters, which was visible in the average execution time even with only 1,000 updated records. Not to discredit the mentioned rule through numerous characters concatenation, the syntax has been adapted by dividing the INSERT ALL into 1,000 records chunks. An INSERT ALL was performed every 1,000 records in as many iterations as needed to insert all records. The mentioned fixed cost of statement concatenation is

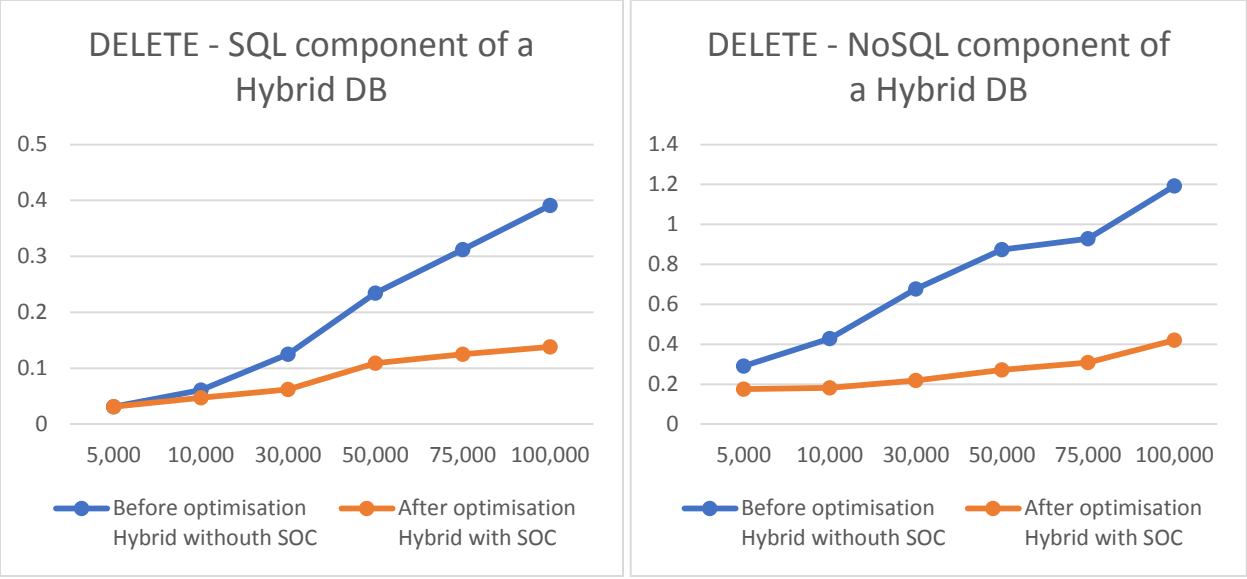
multiplied. However, the average execution time over a larger amount of data highlighted the benefits of executing one INSERT ALL over 1,000 records.



(a) *(b)*

Figure 5 – The average measured execution times of use cases UC_3 (a) and UC_4 (b)

Figure 5 shows the average UC_3 and UC_4 use cases' execution times. The UPDATE statement before optimisation in the destination SQL component averaged the following durations: 0.981 (100,000 records), 2.679 (300,000 records), 4.849 (500,000 records), 8.032 (750,000 records) and 10.835 (1,000,000 records), expressed in seconds. In the hybrid with SOC, the performance was as follows: 0.843 (100,000 records), 2.156 (300,000 records), 2.766 (500,000 records), 3.513 (750,000 records) and 4.519 (1,000,000 records). The limitation of applying the PARALLEL hint is the inability to guarantee its usage. However, tested UC_3 was using the PARALLEL hint. Even though the slight advantage of using PARALLEL was detected even on 100,000 records, the benefit of the parallel record update was more noticeable on 500,000 records. With the increase in the number of records, the average execution time has decreased compared to the non-optimised execution. This time saving occurred as the consequence of the parallel, instead of sequential, statement execution. Before optimisation, UC_4, on the hybrid without SOC, was executed in 18.649 (100,000 records), 65.005 (300,000 records), 94.751 (500,000 records), 142.677 (750,000 records) and 188.009 (1,000,000 records) seconds. Following the optimisation, UC_4 achieved drastically decreased average execution times. The average times for UC_4 after optimisation are 1.473 (100,000 records), 3.381 (300,000 records), 5.054 (500,000 records), 7.907 (750,000 records) and 10.035 (1,000,000 records) in seconds. By far, the greatest absolute and relative savings in average execution time was achieved by the optimised UC_4. The reason for that is the powerful updateMany() mechanism, which significantly comes to the fore in contrast to a million executions of the updateOne() method.



(a) *(b)*

Figure 6 – The average measured execution times of use cases UC_5 (a) and UC_6 (b)

Figure 6 depicts the optimisation effects of UC_5 and UC_6 on the average statement execution times. The deletion of all records in the SQL component was carried out in 0.031 (5,000 records), 0.061 (10,000 records), 0.125 (30,000 records), 0.234 (50,000 records), 0.312 (75,000 records) and 0.391 (100,000 records) seconds. The optimised statement needed 0.031 (5,000 records), 0.047 (10,000 records), 0.062 (30,000 records), 0.109 (50,000 records), 0.125 (75,000 records) and 0.138 (100,000 records) seconds. The identical average time of record deletion, before and after optimisation, over the dataset of 5,000 records showed the efficiency of the DELETE statement when the WHERE clause was not forwarded. However, with the increase in the number of records for deletion, especially over 30,000 records and more, the advantage of using the DDL statement, which does not go through individual rows, came into the spotlight.

The deleteMany operation(), which already represents an improvement over the basic deleteOne() method, needed 0.291 (5,000 records), 0.428 (10,000 records), 0.677 (30,000 records), 0.874 (50,000 records), 0.929 (75,000 records) and 1.192 (100,000 records) seconds and it represents the UC_6 performance before optimisation. The optimised UC_6 took, on average, 0.176 (5,000 records), 0.181 (10,000 records), 0.219 (30,000 records), 0.271 (50,000 records), 0.309 (75,000 records) and 0.421 (100,000 records) seconds. Even though less drastically, the optimised DROP over the MongoDB component also led to performance improvement, expressed through the average execution times.

7. Conclusions and future work

The research in this paper was inspired by the wish to analyse and point to the possibilities and justification for extending the hybrid SQL/NoSQL database by introducing the statement optimisation component. The paper describes the dedicated SOC component and how it interacts with other components of a hybrid database, as well as with the newly introduced SQL repository for storing optimisation rules. A dedicated process model specifies the process of applying optimisation techniques, and it is presented with the UML activity diagram. The illustrated process contains a certain level of

generalisation. The result of this characteristic is a straightforward extension of the optimisation techniques for the existent and additional database types that make hybrid components. On the other hand, the process model is specific enough that it is adjusted for the hybrid SQL/NoSQL databases.

Without striving to cover all possible optimisation rules for all types of databases, which would be an almost impossible task at the moment, demonstrative rules were chosen for optimising INSERT, UPDATE and DELETE statements. Test use cases demonstrate the effects of applying the SOC component. The comparison contains the average durations of the same use cases executed over the hybrid database with and without SOC. The results of the tested use cases contributed towards the conclusions. In some use cases, over certain records, a hybrid database with the SOC component didn't necessarily achieve a shorter execution time (i.e. UC_5 over 5,000 records).

However, the obvious trend observed was that the hybrid database with the SOC component required less time for test statements execution when the number of records increased, in comparison to the hybrid without SOC. The most significant difference in duration was achieved when executing UC_4 over 1,000,000 records. The average statement execution time in the hybrid with SOC was 18 times shorter than the execution of the same UC_4 over the hybrid without SOC. The conclusion that arises is that with the smaller number of records, a hybrid with SOC cannot always necessarily achieve a decrease in the average execution time in comparison to the hybrid without SOC. However, when working with a larger amount of data, experimental tests indicated a significant decrease in the average execution times for the selected use cases of the presented domain. The introduction of SOC requires time and effort for the architecture extension and adjustment for work with the existing components of a hybrid database. Nevertheless, for data-intensive systems that work with a large amount of data, the experiments demonstrated a significant decrease in the average execution times of optimised statements.

In addition to the number of rules for the statements optimisation, which is not final and which can be additionally researched and extended, the particular limitation of the approach from this paper is in the current version of the architecture. The current version still does not support the unified execution of multiple different statements (i.e. UPDATE with subquery or DELETE with subquery). In addition, the DDL statements are not supported in the current hybrid database version. The extension of the present optimisation rules, the introduction of DDL statements, and the support of nested statements of different types represent directions for future research and further hybrid SQL/NoSQL database development.

References

Abdelhedi, F., Brahim, A.A., Atigui, F., Zurfluh, G., 2017. Logical Unified Modeling For NoSQL DataBases. In: 19th International Conference on Enterprise Information Systems (ICEIS 2017). Porto, Portugal, pp. 249-256. ISBN: 978-3-319-93375-7.

Aleem, H.N., Baig, M.M., & Khan, M.M., 2019. Efficient Software Testing Technique based on Hybrid Database Approach. *International Journal of Advanced Computer Science and Applications*, 10 (7), 349–356. http://dx.doi.org/10.14569/IJACSA.2019.0100748.

Bender, B., Bertheau, C., Körppen, T., Lauppe, H., Gronau, N., 2022. A proposal for future data organization in enterprise systems—an analysis of established database approaches. *Information Systems and e-Business Management*, 20 (2022), 441–494. https://doi.org/10.1007/s10257-022-00555-6.

Bjeladinovic, S., 2018. A fresh approach for hybrid SQL/NoSQL database design based on data structuredness. *Enterprise Information Systems*, 12 (8–9), 1202–1220. https://doi.org/10.1080/17517575.2018.1446102.

Bjeladinovic, S., Marjanovic, Z., Babarogic, S., 2020. A proposal of architecture for integration and uniform use of hybrid SQL/NoSQL database components. *Journal of Systems and Software,* 168 (2020), 110633. https://doi.org/10.1016/j.jss.2020.110633.

Breß, S., Schallehn, E., Geist, I. 2013. Towards Optimization of Hybrid CPU/GPU Query Plans in Database Systems. In: Pechenizkiy, M., Wojciechowski, M. (Eds.), New Trends in Databases and Information Systems. Advances in Intelligent Systems and Computing, vol 185. Springer, Berlin, Heidelberg, pp. 27-35. https://doi.org/10.1007/978-3-642-32518-2_3.

Čerešňák, R., Kvet, M., 2019. Comparison of query performance in relational a non-relation databases. *Transportation Research Procedia*, 40, 170-177. https://doi.org/10.1016/j.trpro.2019.07.027.

Cremer, S., Bagein, M., Mahmoudi, S., Manneback, P., 2017. Improving Performances of an Embedded Relational Database Management System with a Hybrid CPU/GPU Processing Engine. In: Francalanci, C., Helfert, M. (Eds.), Data Management Technologies and Applications. DATA 2016. Communications in Computer and Information Science, vol 737. Springer, Cham, pp. 160-177. https://doi.org/10.1007/978-3-319-62911-7_9.

de la Vega, A., García-Saiz, D., Blanco, C., Zorrilla, M., Sánchez, P., 2018. Mortadelo: A Model-Driven Framework for NoSQL Database Design. In: Abdelwahed, E., Bellatreche, L., Golfarelli, M., Méry, D., Ordonez, C. (Eds.), Model and Data Engineering (MEDI 2018), Lecture Notes in Computer Science, vol 11163. Springer, Cham. https://doi.org/10.1007/978-3-030-00856-7_3.

Faraj, A., Rashid, B., Shareef, T., 2014. Comparative study of relational and nonrelations database performances using Oracle and MongoDB systems. *International Journal of Computer Engineering & Technology (IJCET)*, 5(11), 11-22.

Fraczek, K., Plechawska-Wojcik, M., 2017. Comparative Analysis of Relational and Non-relational Databases in the Context of Performance in Web Applications. In: Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., Kostrzewa, D. (Eds.), Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation. BDAS 2017. Communications in Computer and Information Science, vol 716. Springer, Cham. https://doi.org/10.1007/978-3-319-58274-0_13.

Gowanlock, M., Karsin, B., Fink, Z., Wright, J., 2019. Accelerating the Unacceleratable: Hybrid CPU/GPU Algorithms for Memory-Bound Database. In: Proceedings of the 15th International Workshop on Data Management on New HardwareJuly (DaMoN'19), Article No.7. Amsterdam Netherlands, pp. 1-11. https://doi.org/10.1145/3329785.3329926.

Goyal, S., Srivastava, P.P., Kumar, A., 2015. An overview of hybrid databases. In: 2015 International Conference on Green Computing and Internet of Things (ICGCIoT). Greater Noida, India, pp. 285-288. https://doi.org/10.1109/ICGCIoT.2015.7380474.

Gyorodi, C., Gyorodi, R., Sotoc, R., 2015. A Comparative Study of Relational and NonRelational Database Models in a Web- Based Application. *International Journal of Advanced Computer Science and Applications*, 6 (10), 78-83. https://doi.org/10.14569/IJACSA.2015.061111.

James, B., Asagba, P.O., 2017. Hybrid Database System for Big Data Storage and Management. *International Journal of Computer Science, Engineering and Applications (IJCSEA),* 7(3/4), 15-27. https://doi.org/10.5121/ijcsea.2017.7402.

Jatana, N., Puri, S., Ahuja, M., Kathuria, I., Gosain, D., 2012. A Survey and Comparison of Relational and Non-Relational Database. *International Journal of Engineering Research & Technology*, 1(6), 1-5.

Kalayda, A., 2021. Promising directions for the development of modern databases. *Journal of Physics: Conference Series*, Vol. 2131, 022087, 1-6. https://doi.org/10.1088/1742-6596/2131/2/022087.

Kemper, A., Neumann, T., 2011. One Size Fits all, Again! The Architecture of the Hybrid OLTP&OLAP Database Management System HyPer. In: Castellanos, M., Dayal, U., Markl, V. (Eds.), Enabling Real-Time Business Intelligence (BIRTE 2010), Lecture Notes in Business Information Processing, vol 84. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-22970-1_2.

Khan, W., Ahmad, W., Luo, B., Ahmed, E., 2019. SQL Database with physical database tuning technique and NoSQL graph database comparisons. In: IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC 2019). Chengdu, China, pp. 110-116. https://doi.org/10.1109/ITNEC.2019.8729264.

Li, C., Gu, J., 2018. An integration approach of hybrid databases based on SQL in cloud computing environment. *Software: Practice and Experience*. 49 (3), 11–16. https://doi.org/10.1002/spe.2666.

Liu, Z.H., Hammerschmidt, B., McMahon, D., Liu, Y., Chang, H.J., 2016. Closing the functional and Performance Gap between SQL and NoSQL. In: Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16). San Francisco, USA, pp. 227-238. https://doi.org/10.1145/2882903.2903731.

Martinez-Mosquera, D., Navarrete, R., Lujan-Mora, S., 2020. Modeling and Management Big Data in Databases—A Systematic Literature Review. *Sustainability*, 12(2): 634. https://doi.org/10.3390/su12020634.

Mershad, K., Hamieh, A., 2021. SDMS: smart database management system for accessing heterogeneous databases. *International Journal of Intelligent Information and Database Systems*, 14(2), 115-152, https://doi.org/10.1504/IJIIDS.2021.114513.

Nance, C., Losser, T., Iype, R., Harmon, G., 2013. NoSQL vs RDBMS -why there is room for both. In: Proceedings of the Southern Association for Information Systems Conference, pp. 111-116.

Owaida, M., Sidler, D., Kara, K., Alonso, G., 2017. Centaur: A Framework for Hybrid CPU-FPGA Databases. In: IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2017). Napa, CA, USA, pp. 211-218. https://doi.org/10.1109/FCCM.2017.37.

Pang, Z., Wu, S., Huang, H., Hong, Z., Xie, 2021 . AQUA+: Query Optimization for Hybrid Database-MapReduce System. *Knowledge and Information Systems*, 63(2021), 905–938. https://doi.org/10.1007/s10115-020-01542-4.

Roy-Hubara, N., Sturm, A., 2020. Design methods for the new database era: a systematic literature review. *Software and Systems Modeling*, 19 (2020), 297-312. https://doi.org/10.1007/s10270-019-00739-8.

Scheuermann, B., 2010. Design of a Reconfigurable Hybrid Database System. In: 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines. IEEE, Charlotte, NC, USA, pp. 247–250. https://doi.org/10.1109/FCCM.2010.44.

Sellami, R., Defude, B., 2018. Complex Queries Optimization and Evaluation over Relational and NoSQL Data Stores in Cloud Environments. *IEEE Transactions on Big Data*, 4(2), 217-230. https://doi.org/10.1109/TBDATA.2017.2719054.

Sokolova, M., Gómezb, F., Borisoglebskayaa, L., 2019. Migration from an SQL to a hybrid SQL/NoSQL data model. *Journal of Management Analytics,* 7(1), 1-11. https://doi.org/10.1080/23270012.2019.1700401.

SolidIT, 2023a. DB-engines ranking.
Available on: https://db-engines.com/en/ranking (Retrieved: January 2023).

SolidIT, 2023b. DB-engines ranking – Relational DBMS.
Available on: https://db-engines.com/en/ranking/relational+dbms (Retrieved: January 2023).

SolidIT, 2023c. DB-engines ranking – Document store DBMS.
Available on: https://db-engines.com/en/ranking/document+store (Retrieved: January 2023).

Sudhakar, K., 2018. Difference between SQL and NoSQL Databases. *International Journal of Management, IT and Engineering*, 8(6), 444-452.

Thiry, L., Zhao, H., Hassenforder, M., 2018. Categories for (Big) Data models and optimization. *Journal of Big Data* 5, Article No. 21. https://doi.org/10.1186/s40537-018-0132-9.

Villari, M., Celesti, A., Giacobbe, M., Fazio, M., 2016. Enriched E-R model to design hybrid database for big data solutions. In: 2016 IEEE Symposium on Computers and Communication (ISCC). Messina, Italy, pp. 163-166. https://doi.org/10.1109/ISCC.2016.7543733.

Vyawahare, HR., Karde, PP., Thakare, VM., 2018. A Hybrid Database Approach Using Graph and Relational Database. In: 2018 IEEE International Conference on Research in Intelligent and Computing in Engineering. IEEE, Univ Don Bosco, San Salvador, EL SALVADOR, pp. 2555–2564. https://doi.org/10.1109/RICE.2018.8509057.

Vyawahare, HR., Karde, PP., Thakare, VM., 2019. Hybrid Database Model For Efficient Performance. *Procedia Computer Science*, 152 (2019), 172-178. https://doi.org/10.1016/j.procs.2019.05.040.

Zhang, L., Pang, K., Xu, J., Niu, B., 2022. JSON-based control model for SQL and NoSQL data conversion in hybrid cloud database. *Journal of Cloud Computing* 11, Article No. 23, (2022). https://doi.org/10.1186/s13677-022-00302-9.

Zečević, I., Bjeljac, P., Perišić, B., Stankovski, S., Venus, D., Ostojić, G., 2018. Model driven development of hybrid databases using lightweight metamodel extensions. *Enterprise Information Systems*, 12 (8–9), 1221–1238. https://doi.org/10.1080/17517575.2018.1445295.

Figure 1

Click here to access/download;Figure(s);Figure 1.jpg

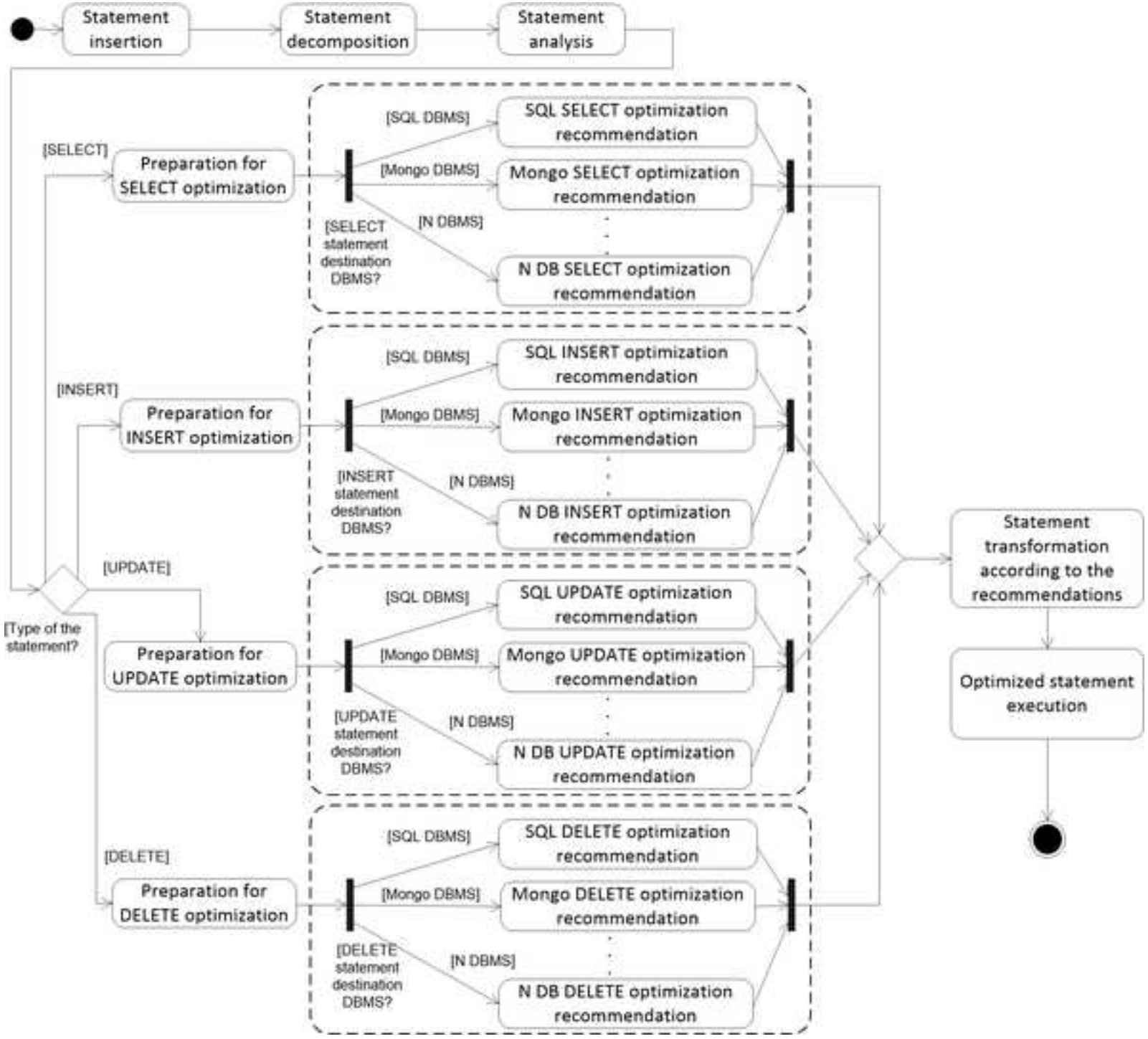


Figure 2

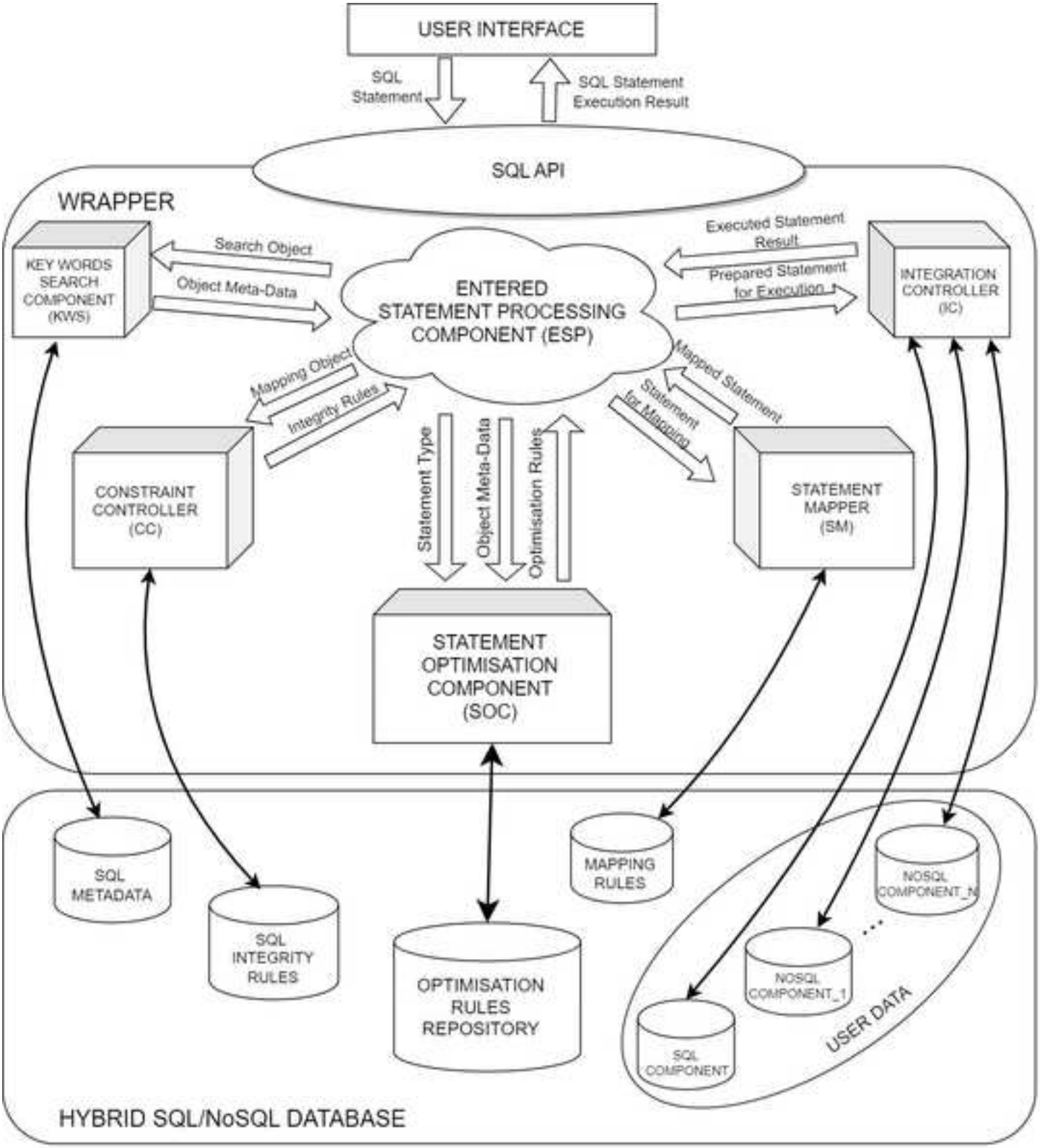
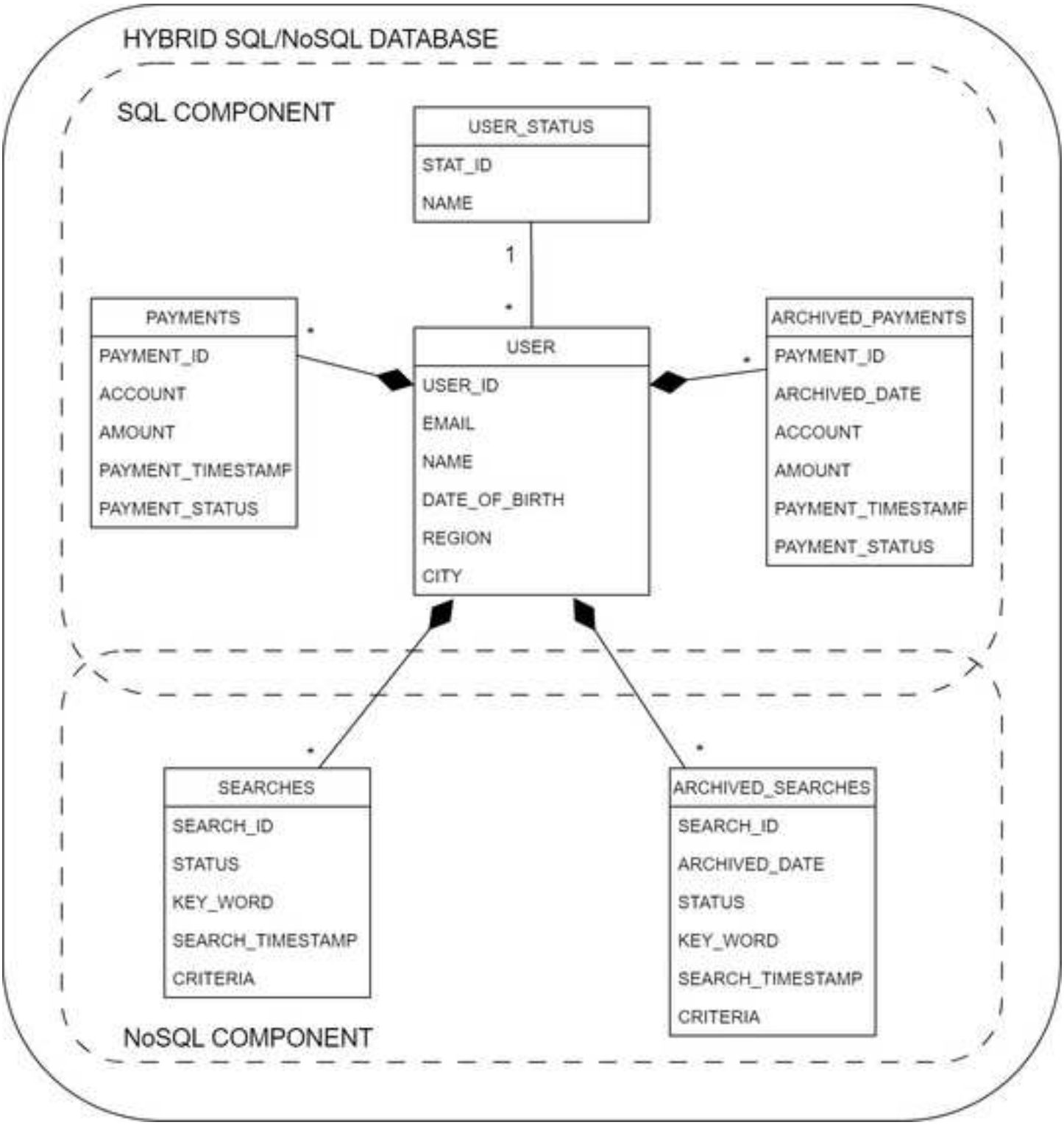


Figure 3



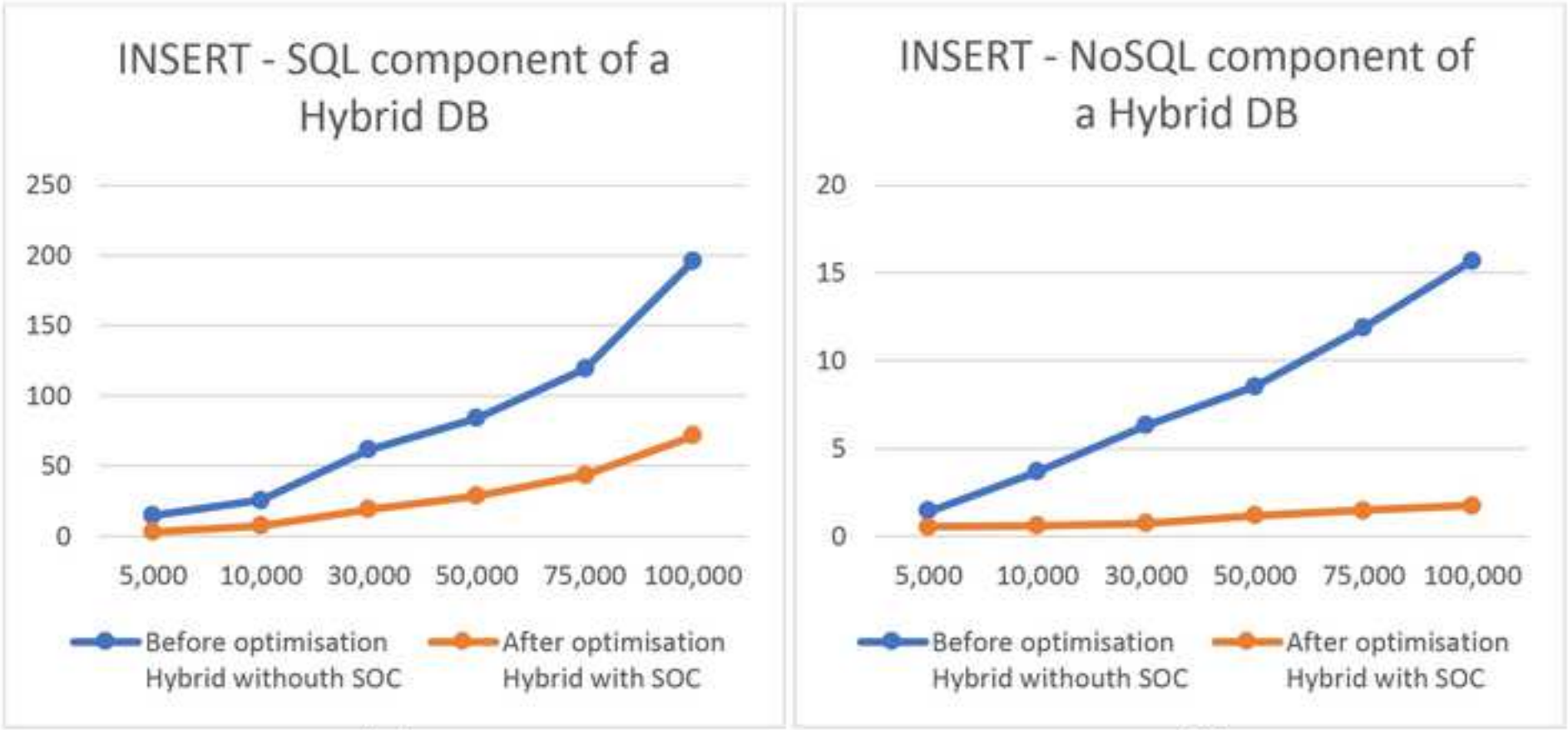
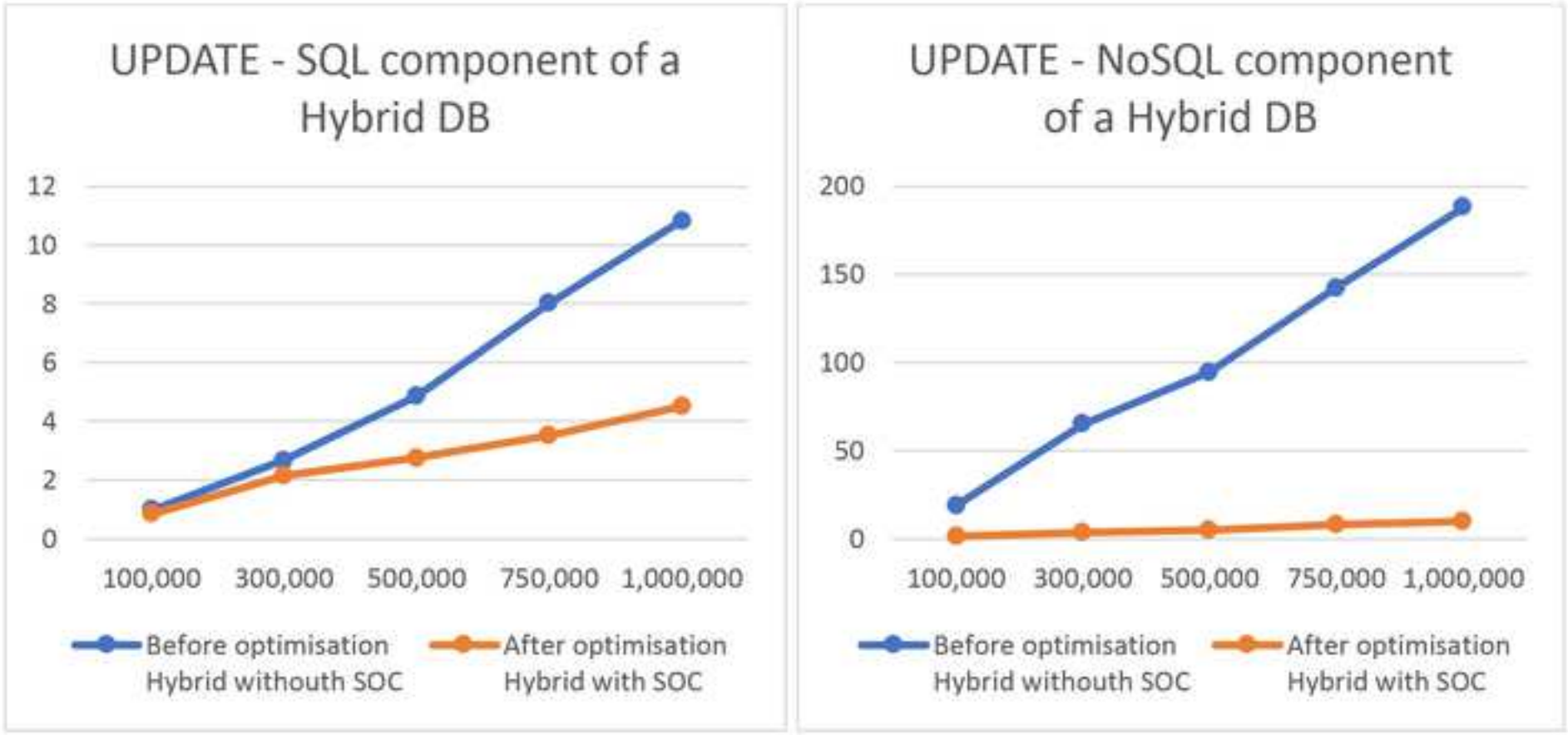


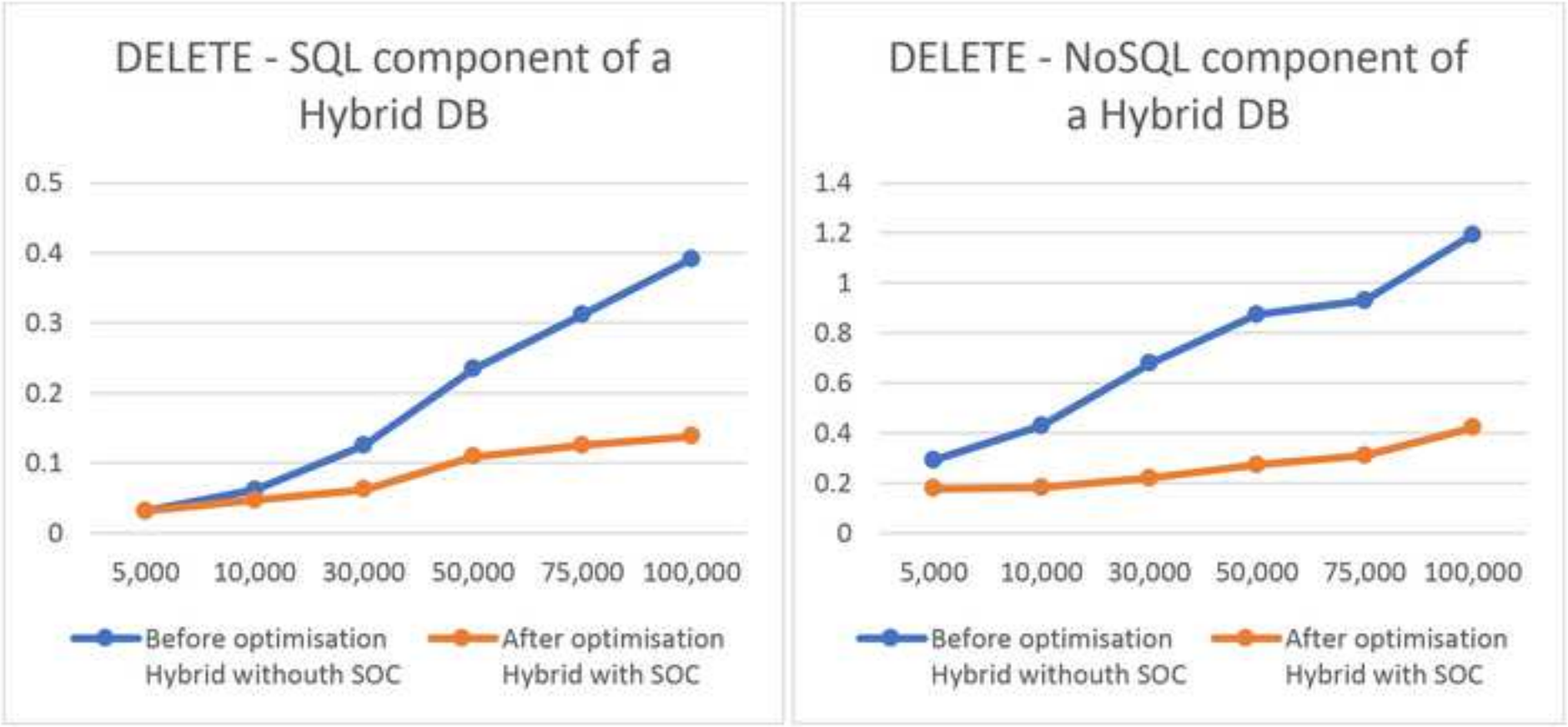
Figure 4 – The average measured execution times of use cases UC_1 (a) and UC_2 (b)

Figure 5



(a) *(b)*

Figure 5 – The average measured execution times of use cases UC_3 (a) and UC_4 (b)



(a) *(b)*

Figure 6 – The average measured execution times of use cases UC_5 (a) and UC_6 (b)