Computer Science and Information Systems 22(3):1011–1046 https://doi.org/10.2298/CSIS241024034B

Extending Hybrid SQL/NoSQL Database by Introducing Statement Rewriting Component

Srđa Bjeladinović

University of Belgrade, Faculty of Organizational Sciences
Jove Ilića 154, Beograd, 11000, Belgrade, Serbia
{srdja.bjeladinovic}@fon.bg.ac.rs

Abstract. Contemporary organisations often include different business 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 unique logical database. At the same time, they provide the usage benefits of working with the SQL and the NoSQL databases simultaneously. Recently, there has been an increase in research that deals with the challenges of hybrid databases' query optimisation, especially query rewriting. This trend opened up possibilities for analysing the influence of applying different statement rewriting techniques to other data manipulation statements besides queries (i.e. INSERT, UPDATE and DELETE) and its impact on the average execution times. In this paper, a process model for applying automatic hybrid statements' rewriting was designed, and the architecture for the hybrid database was extended with the newly developed Statement Rewriting Component (SRC). The tested use cases were conducted on the example of Oracle/MongoDB/Cassandra hybrid before and after introducing SRC. The tests have shown particular decreases in the average execution times of the system with the SRC.

Keywords: Hybrid database; SQL; NoSQL; statement rewriting; database architecture.

1. Introduction

Since the beginning of the 90s of the last century, it has been identified that individual systems cannot always satisfy all business needs and that it is often necessary to design, manage and maintain systems made up of several other systems. The development of complex systems, whose components are intensive software systems, usually developed by different manufacturers, was significantly influenced by the rapid growth of society and industry [1]. The constant increase of the business processes complexity, the expansion of the portfolio of products and services, the diversification of business between companies and the expansion of software specialised for specific domains of business or industrial branches have further promoted the development of systems that integrate other, independent, systems. Individual and independent systems cannot always achieve a higher business mission. Still, their collaboration and integration into complex systems enable the realisation of missions beyond their capabilities [2]. These systems are sometimes referred to as Component-Based Software [3] or Systems-of-Systems (SoS) [4]. Today, large and complex systems that contain other constituent systems [5] are represented in many fields [6][2][7], such as transportation networks, smart energy grids, space,

aeronautics, e-commerce applications, medical assistance, emergency management, and databases.

To eliminate possible terminological ambiguities, the interpretations of common terms used in this paper are defined before the term hybrid database. Generally speaking, two main aspects of databases are data model and DataBase Management System (DBMS). As defined by ANSI [8], *"DBMSs can be categorised in terms of the data model which is supported and the data language provided for interacting with this data model. A data model defines the acceptable types of data structures."* So, data models, from the DBMS point of view, represent an appropriate structure for storing and managing data. The authors [9] define a data model as a theoretical approach that determines the way of specifying and designing a specific database, while a DBMS is a particular data processing technology, i.e. a software system, which allows management of large amount of data and implements a specific data model. Further, data model instances can belong to one of three types:

- Conceptual data model - Corresponds to the conceptual schema of the entire system and describes the domain semantics without technological and implementation specifics;
- Logical data model - Describes the semantics in the context of a specific technology for data manipulation, i.e. specific structures suitable for system development (tables and columns, JSON documents, graphs, etc.). It represents the result of mapping the conceptual data model (conceptual schema) into the schema of a data model supported by a specific DBMS [10];
- Physical data model - A data model corresponds to the design of the internal, physical structure of the database based on the logical model and the specification of all non-functional requirements [9].

A hybrid is a unique logical database designed over a single conceptual data model. The conceptual data model is translated into the logical data models of the particular DBMSs, which are chosen for building the hybrid's components. A hybrid database integrates all logical data models of its components into a unique logical unit via unified processes of designing and administrating hybrid and its components, as well as using an integrated meta-repository.

Depending on the types of integrated DBMSs, logical models contained in a hybrid SQL/NoSQL database can describe tables and columns that implement the relational model, JSON document and its fields, ordered key-value pairs, column families, or graph nodes and their relationships. The hybrid SQL/NoSQL databases provide the benefits of working with the SQL and the NoSQL databases simultaneously. Through the hybrid, each component (DBMS) participates in achieving the broader business mission of the company, which could not be satisfied as an isolated DBMS.

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. This approach in developing data-intensive systems enables the use of suitable DBMS representatives for each business domain or, more precisely, for each business subdomain. Contemporary organisations often unify business functions that use strict and, in advance, defined data structures with other business functions that use highly flexible data structures. Apart from the simultaneous use of data with different levels of structuredness, various criteria may have higher or lower importance, depending

on the business needs. For example, we can observe the information system of a marketing company, which entails monitoring and executing 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, i.e. to use more structured data with the highest level of consistency. In the case of social media promotion, data integrity and structuredness are often secondary-significant to the data availability and fast analytical processing.

Before the emergence of hybrid databases, the challenge of using data with different levels of structuredness was solved in one of the following two approaches: 1) limiting organisations to using an exclusive database type for all subsystems; 2) using different types of databases for each of the subsystems (or similar subsystems). In the previous example of a marketing company, as in many other examples of business, it is not always easy nor optimal to choose a single database type. The usage of exclusive database type gives the organisation benefits for some specific subsystems. At the same time, other subsystems of that company have the limitations of using a non-suitable or non-optimal database type. Because of that, the first legacy approach cannot resolve all specific demands raised by the necessity of using different types of databases.

The second approach to the problem, using different types of databases (with varying data models) whose operation is not fluidly represented as the unique logical database, implies additional expenses of connections, integration and unified administration. Compared to the two prior approaches (single database and multiple databases over multiple models), a hybrid system of databases enables integration and concurrent use of different technologies. Hybrid database users benefit from using the best functionalities of different database types. Users of hybrid do not have to worry about their integration since different types of databases represent the components of the unique logical (hybrid) database, which uses a single data model. Hybrid database usage enables the engagement of suitable data storage and management technology for every business subdomain. The most common and comprehensive representatives of the hybrids are the hybrid SQL/NoSQL databases.

Even though, in recent years, there has been a noticeable increase in the number of works and research on hybrid database optimisation and statement rewriting, additional space for research has emerged. For the optimisation of databases, in general, different approaches can be used, such as logical optimisation, physical optimisation (e.g. horizontal and vertical partitioning), various ways to access data (e.g. via tables, indexes, materialised views) and reformulating the way statements are written. The latter contains a potential not fully explored in existing papers.

This manuscript is a continuation of research and expansion of the hybrid SQL/NoSQL database and its architecture presented in previous works [11][12]. This paper aims to give answers to the following research questions which have arisen during the continuation of the research:

- Research question 1: How do we specify the automatic usage of statement rewriting techniques in a hybrid SQL/NoSQL database?
- Research question 2: Is it feasible to develop a new dedicated component for statement rewriting and integrate it into the existing architecture for hybrid SQL/ NoSQL databases?

– Research question 3: How do the applied statement rewriting techniques influence the duration of the statement execution, based on the example of the Oracle/MongoDB/ Cassandra hybrid database?

The answer to the first research question covers the extension of the existing hybrid database design methodology [11], and it introduces the designed process model and activities for rewriting entered statements (particularly INSERT, UPDATE and DELETE). Automatic statement rewriting is one of the powerful optimisation techniques, and it aims to reduce statements' average execution time of the initial architecture of SQL/NoSQL database [12] by introducing a newly developed *Statement Rewriting Component (SRC)*. The second question of this paper deals with how to implement SRC on the existing architecture for a hybrid SQL/NoSQL database and how to seamlessly integrate SRC with the execution of other components of a hybrid. The third question aims to quantify conducted practical tests of the selected use cases and compare the achieved results of the existing SQL/NoSQL architecture (without SRC) and extended SQL/NoSQL architecture (with newly developed SRC component).

However, there are some limitations in this paper. The current architecture version of the hybrid SQL/NoSQL database is still a prototype, and not all functionalities are entirely implemented. The SELECT statement is out of the scope of this paper for several reasons. The first one is that the initial architecture version for SQL/NoSQL databases (without SRC) already contains some built-in mechanisms, but only the basic ones, like indexing and partitioning. We acknowledge that query optimisation is thoroughly researched in the analysed papers. Because of that, in this paper, we focus on one particular optimising technique (statement rewriting) of the other data manipulation statements (INSERT, UPDATE and DELETE), which is, to our knowledge, not researched enough. The main limitation of this paper is that we focus exclusively on providing automatic statement rewriting as one of many techniques for statement optimisation, but at the same time, it is a very promising technique considering its potential effect on the statement's average execution time. However, the Optimisation Rules Repository, created and introduced in this paper together with SRC, present a useful base for integration and support not only with the new SRC component but also with future components of the hybrid.

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 automatically applying statement rewriting techniques in hybrid databases. Section 4 describes the architecture extension, with a newly created component for statement rewriting. A description of the SRC component's role in the architecture and an explanation of its functioning aims 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 SRC component and for the hybrid SQL/NoSQL databases with the newly created SRC. The Oracle/MongoDB/Cassandra 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 [13] or as databases that incorporate different types of databases into a unique logical database [11]. By reviewing their presence on the market [14], 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 [15]. For years, big companies such as Facebook, Amazon and Google have been using SQL and NoSQL databases to complement each other [16]. Recent research on how four representatives of NoSQL handle a variety of data is presented in the paper [17]. 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 [18]. At times in literature, the NoSQL databases are called non-relational [19][20][21]. However, non-relational databases can be generally treated as a broader term, including other types of databases [22]. The increased popularity of the NoSQL databases directly affected the focus of databases' hybridisation in recent years. The focus has shifted towards integrating NoSQL and SQL databases into the hybrid SQL/NoSQL database [23]. Therefore, the authors [19] 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 [24] defines the term hybrid database or just the hybrid as "systems where there are several databases implemented that can be relational and/or NoSQL." Based on all of the covered definitions, it is very important to highlight the common trait of a hybrid. Each hybrid database, no matter how many different types of databases it includes, is created over just a single data model and thus is designed and maintained as a unique logical database. Various parts (components) of a hybrid's conceptual data model are implemented in different DBMSs, which can represent very different database types. However, users always access and interact with a hybrid database as with a single database (which implements the entire data model and its necessary logic), no matter which particular hybrid component, e.g., a specific DBMS, stores searched data.

The review in the field of hybrid databases identified four groups of papers of interest. Papers in the first group explain similarities but also crucial differences between hybrid databases and other contemporary databases, which contain multiple systems similar to the SoS. The second group comprises the articles that contribute towards setting hybrid databases' general principles (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 optimisation of various database types represent the third 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 fourth group of analysed papers directly discusses the hybrid database's statements optimisation and rewriting.

2.1. Similar but Different: Alternatives for Hybrid Databases

In recent years, significant progress has been made in the directions that have a tangential research question with hybrids, which is the integration of different systems of databases, i.e. different types of databases. However, due to noticeable differences in the way of

solving the mentioned problem, as well as obvious distinctions in these approaches, it would be risky, even indescribable, to equate them. However, because of their popularity, some of these approaches will be briefly mentioned here. These approaches are polystores, polyglot persistence, multi-model databases, and Object-NoSQL Data Mapper (ONDM) frameworks. Table 1 shows the taxonomy of the approaches from the first group, their key characteristics, similarities and differences compared to a hybrid database.

Table 1. Taxonomy of the approaches dealing with the SQL and NoSQL integration (alternatives to the hybrid databases)

Approach	Key characteristics	Similarities to hybrid DB	Differences to hybrid DB
Polystores	Orchestration of different models and improving the usage of a uniform language on multiple databases	Simultaneous use of several different types of databases; Single query language	Numerous isolated data models, which are subsequently linked; Not the unique logical database
Polyglot persistence	Using different types of databases to solve conflicting requests	Use "the best" type of database for the concrete requests	Absence of a unique language for accessing all databases; Not the unique logical database
Multi-model databases	Ease of use of different models within a single database	Using different models for different requirements; Using one logical database	Only one DBMS; Number of different data models limited by particular DBMS
ONDM	Object model instead of an approach for integrating SQL and NoSQL	A sense of using one logical database	Only one data model (object) instead of a variety of different ones; List similarities with the hybrid databases

Polystores is an approach that deals with the simultaneous use of several different types of databases. They don't have one common data model for the entire logical database, but instead numerous isolated data models, which are subsequently linked. The principles of polystores are described in detail in the paper [25], in which the authors emphasise the importance of using a uniform language over multiple data models. They have presented a BigDAWG prototype composed of SciDB, Accumulo, Postgres and S-Store DBMS. The research mentioned above was extended in subsequent works [26][27][28][29], while the authors [30] have developed purpose-built modelling tool, named TyphonML, which automatically generates CRUD API for polystores. The consequence of the simultaneous, not necessarily related, design and the use of multiple data models is reflected in the ploy-

stores' inability to provide a unique administration process for all databases in use, which implies an additional difficulty in controlling and reducing unwanted data redundancy effectively. The maiden authors of polystores [25] mentioned this as a direction for further research. In addition to the above, synchronisation and simplicity of system expansion, as two of the indicators of integration and usability [31], are aggravating in the case of polystores architecture.

Polyglot persistence represents one of the approaches of using different types of databases to solve conflicting requests, in which only one database cannot solve all tasks. A detailed description of this approach and its specific variants are given in the paper [32]. The authors of the mentioned work identify the following subcategories, which differ in the realisation of the polyglot persistence approach: (I) Application-coordinated Polyglot Persistence (ACPP); (II) Service-oriented Polyglot Persistence (SOPP); (III) Polyglot Persistence as a Service (PPS). Polyglot persistence defines four types of cardinalities between application modules and DBMS: One-to-one, Many-to-one, One-to-many and Many-to-many. The first two cardinalities (1-1; M-1) eliminate the possibility of using different types of databases because they have only one destination database type. The other two cardinalities (1-M; M-M) imply using different types of databases but with the parallel use of multiple query languages, which the authors clearly emphasised in their work [32]. Polyglot persistence with cardinalities 1-1 and M-1 is not of interest to this paper because, with these cardinalities, clients are limited to using only one type of database. On the other hand, cardinalities 1-M and M-M in polyglot persistence approaches, with all their variants (ACPP, SOPP and PPS), allow the use of several different types of databases but require the simultaneous knowledge and use of several query languages. Consequently, the used databases cannot be treated as unique logical databases because it is necessary to separate the data by database types, which results in additional difficulties of integration and reduction of redundancy.

Multi-model databases were created to offer ease of use of different models within a single database. For example, Redis, which is primarily a key-value database, supports document-oriented and graph models, while Elasticsearch supports search engine and document-oriented types of databases [14]. A detailed review of multi-model databases was given by the authors [33]. Authors Lu and Holubová investigated the multi-model database from several aspects: the way of handling a variety of data [34] and the comparison with polystores [35]. Some authors recently compared multi-model databases with polyglot persistence [36][37]. Elaborate analysis of the evolution of multi-model databases and directions for further development are detailed in the paper [38]. The authors Lajam and Mohammed [32] state that despite the support for multiple models in one database, multi-model databases still cannot adequately compete with polystores, i.e. systems made up of several different types of databases, primarily in terms of satisfying non-functional requirements such as scalability and performance. Apart from those mentioned above, a significant limiting factor in comparisson to hybrids is the closedness of each individual DBMS. Although certain DBMSs support several different models, the extension of the set of supported models is strictly locked and completely dependent on the DBMS manufacturer. Vendor lock-in can lead to a model not being implemented in the observed DBMS in the future, either. Those mentioned above noticeably limit the possibilities of designers to compose a database of several models and maintain and expand the set of models used by the user.

ONDM research focuses on mapping and integrating object and NoSQL models and extends the functionalities introduced by Object-relational mapping (ORM) frameworks [39]. However, the primary focus of these frameworks is not the direct integration of SQL and NoSQL types of databases. Instead, to a certain extent, they achieve it via an object model.

Table 2. Pros and cons of the analysed approaches

Approach	Pros	Cons
Polystores	Simultaneous use of different types of databases; Usage of a uniform language	The inability to provide a unique administration process for all databases; Difficulty in controlling and reducing data redundancy; Synchronisation and simplicity of system expansion
Polyglot persistence	The use of several different types of databases	Use of several query languages; Difficulties of integration and reduction of redundancy
Multi-model databases	Suported diferrent models; One logical database; The convenience of accessing and managing only one DBMS	Limitations of (broaden) usage of different models in a single DBMS; Vendor lock-in and closedness of each individual DBMS; It is harder to achieve non-functional requirements such as scalability and performance
ONDM	A single data model (object) tries to “fit all”	Not many authors treat this approach as a genuine alternative to a hybrid; Only one data model (object) with its limitations instead of a variety of different ones

Table 2 summarises the pros and cons of the four analysed approaches. The conclusion that can be drawn is that each approach, of course, has advantages and disadvantages, the same as the hybrid databases. Nevertheless, it can be unequivocally concluded that, despite their benefits, the four analysed approaches cannot be treated as adequate alternatives to hybrid databases. The reason is either because of the problem they are trying to solve or because of the way they approach the solution. In particular, although polystores support the parallel use of different types of databases using a unique language, the absence of the possibility of designing the entire database as a unique logical whole without unwanted redundancy makes it impossible to equalise polystores and hybrids. The main difference between the polyglot persistence approach and the hybrids is reflected in not supporting a single language for working with all types of databases. Individual multi-model databases and ONDM show significant disadvantages compared to the hybrids,

which result from the limits of a single DBMS usage, i.e. single data model usage (object model).

2.2. General Principles of Hybrid Databases

The second group of papers deals with hybrid databases' general principles. The authors [40][41][42] 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 the unique logical entity. The authors [43] dealt with developing dedicated design methodologies for hybrid databases. The authors [44] analysed software test techniques for the hybrid databases. The authors [23] highlighted the use benefits of the hybrid SQL/NoSQL databases and presented dedicated concepts 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 [13][45]. In their paper [46], the authors focused on the diversity of NoSQL models (key-value, document-oriented, column family, graph). They highlighted that relational databases' traditional design approaches cannot be directly applied to NoSQL design. They presented the Mortadelo framework based on the model-driven transformation process. It transforms the generic 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 [47] presented the migration approach from the SQL into the hybrid SQL/NoSQL database using ontology to define data schema. In their paper, the authors [48] presented how a conceptual data 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 [48] created a generic logical layer suitable for work with three types of NoSQL databases (document-oriented, column family, and graph). By applying the QVT (Query-View-Transformation) rules, this layer enables efficient transformation execution into the physical model, decreasing the influence of the destination NoSQL database's technical specifications. The authors [43] 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 data 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 [49] 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. Also, authors [50][51] researched some of the challenges of working with heterogeneous databases. The authors [50] have emphasised differences in the consistency and transaction limitations between SQL and NoSQL DBMS.

Furthermore, they have developed a comprehensive approach for managing distributed transactions with guaranteed ACID in heterogeneous data store environments, regardless of whether the individual data stores support ACID. The authors [51] focused on the challenge of mapping syntactically and semantically related attributes among schemas. The

heterogeneous data stores used as the target databases can also be useful for this research because they can present the components of the hybrid databases.

2.3. Database Optimisation and Performance Measurement

The third group of papers deals with different aspects of performance measurements and optimisation of different database types. The authors [52][53] compared query execution in relational and non-relational databases. The paper [53] compared the three DBMS's login and usage performances (PostgreSQL, MongoDB and Cassandra), while the article [52] 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 authors have developed and used a data model of train stations and stops. They have concluded that for Slovakia, it is justified to use an SQL database for storing and managing data. In contrast, for countries with a larger amount of train data, such as the Netherlands or Germany, it was suggested the usage of a NoSQL database. The authors [54] have decreased the performance gap between the SQL and NoSQL databases 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. Detailed analysis of the evolution of using different JSON functionalities, in both native and binary formats, and their influence on the performance of Oracle DBMSs was given in the paper [55]. The authors Kemper and Neumann [56] 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 [56]. From the theoretical aspect, the authors [57] dealt with optimising 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 [58] 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 optimised the execution plan considering different execution engine types. Also, the paper focuses on the execution plan adjustment to the hardware components without considering hybrid SQL/NoSQL database optimisation specificities. The authors [59] 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

[60][61][62]. Even though, in general, the mentioned papers [60][61][62] discuss the optimisation of hybrid systems and hybrid databases, they will not be further analysed in this paper, given their focus on the hardware optimisation.

2.4. Hybrid Database's Statements Optimisation and Rewriting

The fourth group of papers deals with hybrid database statements optimisation and rewriting. The authors [63] dealt with hybrid optimisation 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), intending to reduce the amount of data needed to be processed by the MapReduce hybrid component. The authors [64] 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 heterogeneous databases was achieved by the authors [65]. 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. Li and Gu [66]) developed a useful solution to the nested query optimisation problem over the hybrid operation. In the mentioned paper, they solved 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 the 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"*[66]. Even though it is stated that the presented architecture supports optimisation, the article focuses only on one optimisation segment, and that is the nested queries.

The analysed papers contributed to the purposefulness of further research on the statement rewriting in hybrid databases and its following effects on the architecture changes. It opened up possibilities for and motivated authors to expand further the research of statement rewriting not on queries (because they have been too frequently researched) but on other statements (INSERT, UPDATE and DELETE) in the hybrid databases, which, according to our knowledge, are not explored in detail so far.

3. Process Model for Statement Rewriting 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 other statements enhancing (INSERT, UPDATE and DELETE), and especially for researching the effects of the statements rewriting on the average execution time of a hybrid SQL/NoSQL database.

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 uniform 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 the automatic application of statement rewriting (as an optimisation support technique) for all three statements mentioned above, a dedicated process model was designed. Figure 1 depicts the UML activity diagram for the statement rewriting 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 the integrated conceptual data model (common for all hybrid components), and the user disburden of knowing what component stores needed data. 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 exclusive language for the statement input over the hybrid database, and that language is the standardised SQL. 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 can access all the data as if stored in a single relational database. The user uses only 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 rewriting, which enables the stated operation, is given in Section 4.

The preparation activity of a statement rewriting starts after analysing and decomposing the entered 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 future work.

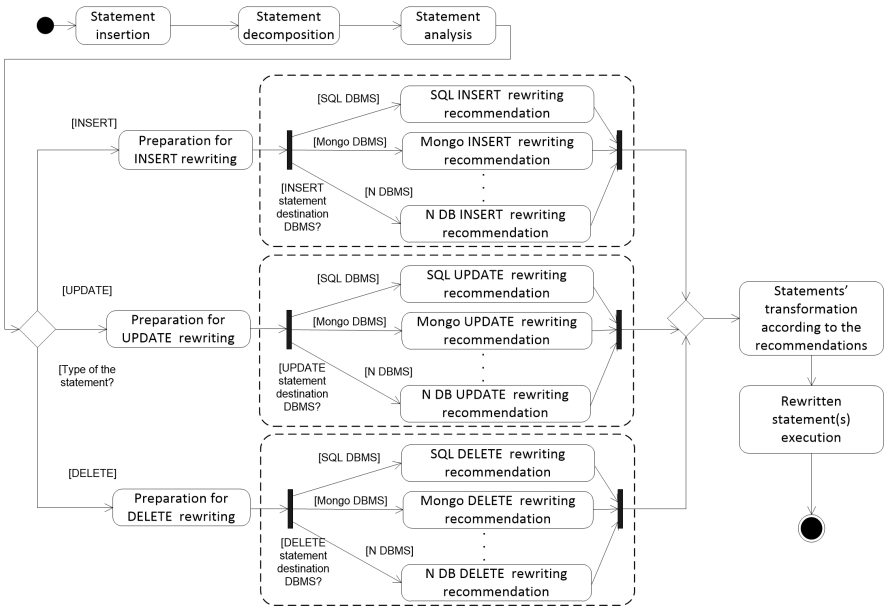

Fig. 1. The UML activity diagram for automatically applying recommendations for statement rewriting on the SQL and NoSQL components of the hybrid SQL/NoSQL database

The statement rewriting 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.

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 rewriting of different parts of the input statement. These parts could be executed into various destination components of the hybrid database.

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

statement rewriting. If necessary, the keywords of the entered SQL statement are mapped according to the language of the destination database, i.e. the 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, a prepared and (according to the accepted recommendations) rewritten statement is then executed. Statement execution represents the last activity on the diagram depicted in Figure 1.

The presented diagram displays the generic logic of introducing the statement rewriting over different components of a hybrid database. A hybrid Oracle/MongoDB/Cassandra database is the chosen representative to depict some of the supported statement rewriting use cases enabled by this approach. As the name suggests, this hybrid consists of three components: Oracle DBMS, the representative of the relational DBMS; MongoDB, the representative of the document-oriented DBMS; and Apache Cassandra, the representative of the wide-column 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 [67], MongoDB was the most popular document-oriented and NoSQL system overall [68], while Apache Cassandra was the most popular wide-column DBMS and the third most popular NoSQL system [69]. Although, from the technical point of view, there is a possibility to have multiple SQL components, the core idea behind a hybrid is to use the most suitable database type (i.e. hybrid component) for a particular organisational subdomain and a particular set of business requests. Because of that, we don't find it appropriate to introduce more than one SQL DBMS but intentionally use the chosen one SQL DBMS for all requests that the SQL database type should cover. However, the possibility of switching to another SQL system is supported. That scenario boils down to the migration process from one database to another (for example, from Oracle to SQL Server, from MySQL to PostgreSQL, etc.). However, the migration between different SQL databases is not in the scope of this paper.

On the other hand, to demonstrate the usage of different NoSQL subtypes (particularly document-oriented and wide-column), two NoSQL components are present in the hybrid database, which was developed to practically test the architecture before and after introducing SRC. Expanding the test database with even more additional subtypes would be useful, but that is planned for upcoming research. The main reasons for that are the existing limitations of the test environment and the complexity of the parallel introduction of additional components on the old architecture without SRC and the improved architecture with SRC, which exceeds the extent of the conducted research.

Section 6 consists of the average execution time for each tested use case. The supported recommendations for certain statements rewriting, classified into Oracle/ MongoDB/Cassandra hybrid database components, are given in Table 3. These recommendations represent supported statement transformation techniques in the current version of the system. The list of the supported rewriting rules is extendible and will be expanded in future research.

The process model in Figure 1 is comprehensive, and it depicts rewriting techniques' application activities for all DML statements. Even the first version of the hybrid SQL/ NoSQL database incorporated basic optimisation techniques for the SELECT statement (query writing syntax, indexes, partitioning, etc.). The optimised SELECT is already shown through the test queries' execution results in earlier papers [11][12]. For this reason, the focus of this paper is on the rewriting of other DML statements (INSERT, UP-

DATE, DELETE) through the application of the newly developed SRC component, as well as on the effects of stated rules usage on chosen use cases.

Table 3. Supported statement rewriting techniques for the hybrid Oracle/MongoDB/Cassandra database

Type of statement	Statement scenario	Recommendations for statement rewriting: *SQL component - Oracle*	Recommendations for statement rewriting: *NoSQL component - MongoDB*	Recommendations for statement rewriting: *NoSQL component - Cassandra*
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*	BATCH INSERT syntax; *The decrease in the number of calls to the database, compared with the multiple calls of the individual INSERT method*
UPDATE	Updating multiple rows	PARALLEL update; *Under certain conditions, hint PARALLEL enables the rewriting via parallel execution of the UPDATE statements instead of the default sequential execution*	*updateMany()* method; *Update multiple documents that satisfy the filter specified as the first argument instead of executing individual updateOne() methods numerous times*	BATCH UPDATE syntax; *The decrease in the number of calls to the database, compared with the multiple calls of the individual UPDATE method*
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*	TRUNCATE statement; *Using the TRUNCATE statement gives the benefits of quickly deleting a large amount of data while preserving the table structure. It is a similar method to the SQL component (Oracle)*

For the rewriting of the INSERT statement, supported syntaxes are INSERT ALL (for the Oracle SQL component), BULK INSERT (for the MongoDB component) and BATCH INSERT (for the Cassandra component). The principle is the same, and the expected benefit is in the shortened average execution time of the rewritten statement due to the one call to the database, compared to the multiple calls for the execution of many individual

INSERT statements (before rewriting). This principle represents the essence of the enhancement regardless of whether INSERT ALL (Oracle), BULK INSERT (MongoDB) or BATCH INSERT (Cassandra) are used.

The UPDATE statement rewriting 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. The limitations of Cassandra's UPDATE statement are reflected in the obligatory WHERE clause, which must contain all the primary key fields. Besides, the IN operator usage is not supported in conjunction with the primary key fields. As a result, Cassandra does not support multiple target row selection within a single UPDATE statement. For the Cassandra component of a hybrid, the applied recommendation is BATCH UPDATE. The BATCH syntax is not much more complex than multiple UPDATE execution, but it reduces the number of calls to the database (one versus numerous). Each input statement should satisfy preconditions for the rewriting 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 3 shows a more efficient 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). The TRUNCATE recommendation applies to Oracle and Cassandra components of the used test hybrid database. Applying the TRUNCATE statement is additionally powerful with the Cassandra NoSQL component. In Cassandra, it is not feasible to execute delete from a table without the WHERE clause (in contrast to SQL databases). Because the WHERE clause with the primary key is mandatory, every deletion of all records requires a preceding SELECT statement to get the IDs of all records. In contrast to that, the TRUNCATE statement is executed without preceding SELECT. In the described case of data deletion in MongoDB, the *drop()* method can be used. 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()* operation.

Table 3 shows implemented suggestions for specific statement rewriting within the identified use cases for working with multiple records at once. The presented recommendations are specified in the syntax of three components of a prototype hybrid database (Oracle/MongoDB/Cassandra), purposely built to test the new architecture. The list of recommendations cannot be treated as final. Table 3 presents the rules implemented so far for the syntax optimisation of the entered statements using the rewriting technique, which is automatically performed by the SRC component of the new architecture. Besides that, the scenario of using different SQL DBMS as the SQL component of a hybrid requires additional effort to specify and implement recommendations syntactically adapted to the chosen DBMS. For instance, although the TRUNCATE command is supported by other

popular SQL DBMS systems (such as MS SQL Server, PostgreSQL, MySQL, etc.), we are aware that this is not the case with all the recommendations given.

Another example is that the MS SQL Server does not support the syntax of BULK/ BATCH INSERT, unlike Oracle and Cassandra. Instead, it allows specifying values of the multiple new records in parentheses after the VALUES clause. Similarly, although PostgreSQL does not explicitly support the BULK/BATCH UPDATE syntax, it adds a FROM clause to the UPDATE statement to achieve the same effect.

It is important to note that the supported statement rewriting techniques cannot always apply. The focus is on 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 rewriting rule, as shown in Table 4.

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 example, in the selected SQL component (Oracle) for applying PARALLEL onto the UPDATE statement (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 generic 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 inadequate 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 rewriting 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 SRC component uses the optimisation advantages of particular DBMSs, with limitations within those DBMSs. The described condition is evident when individual DBMSs are observed, especially within hybrid databases. If the provided hint in the SQL database has a higher execution cost value than its alternatives, the execution engine will not use it. 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. 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 rewriting techniques for the entire 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 SRC components that operate under the process model presented in this Section to improve performance.

4. Extending Hybrid SQL/NoSQL Database with Statement Rewriting Component (SRC)

The architecture presented in the paper [12] 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 unique 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, more precisely, statement rewriting. This paper presents the extension of the initial architecture with the design of the SRC component. Figure 2 shows the extended SQL/NoSQL database architecture with the SRC component, which implements the principles of the newly introduced process model in Figure 1.

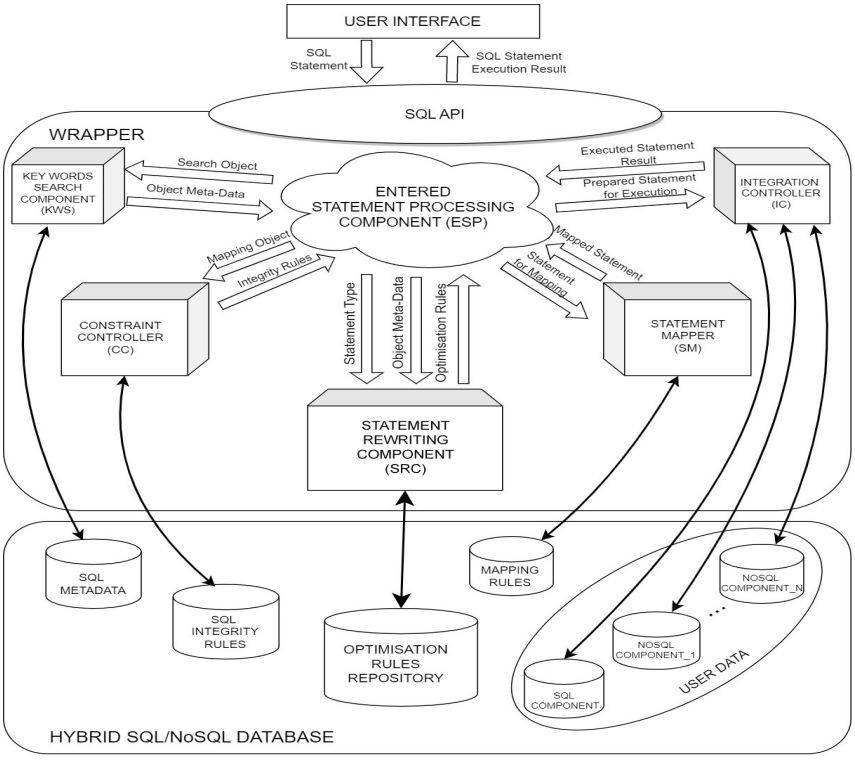

Fig. 2. The SQL/NoSQL database architecture extended with the newly developed component for the statement rewriting (SRC)

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.

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 the Entered Statement Processing Component (ESP), the central communication component of the Wrapper. To achieve this, ESP 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 ESP component is most similar to the traditional three-tier architecture controller. The components ESP 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 Rewriting Component (SRC).

After analysing the SQL statement entered by a user, the ESP component communicates with the KWS component by sending the objects' names (read from the appropriate 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 the unique logical database. Given that all hybrid objects, regardless of what component they belong to, are integrated with the joint data model, 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 ESP 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 an 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 the 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 [70], 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. It 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 Repository 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, which is 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 SRC component. The earlier version didn't contain the SRC 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 SRC component before interacting with the SM component. The ESP component sends the statement type and metadata of destination objects to the SRC component. The SRC 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 (in this case, for statement rewriting).

Table 4 shows the structure of the Optimisation Rules Repository. This table contains selected examples of the statement rewriting rules for INSERT, UPDATE and DELETE of the hybrid Oracle/MongoDB/Cassandra database.

If the need for introducing new rules occurs, the new record will be added to the Optimisation Rules Repository. One record will be added for each statement type of each existing database. Additionally, if the hybrid database expends to the additional components (databases), new rules, in the form of new records in the Optimisation Rules Repository, will be 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 columns with preconditions. Each precondition corresponds to a column of the same name. In Table 4, optimisation rules, in this case rewriting 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 4. Columns whose headings are preconditions' names represent a specific optimisation technique known at Oracle under 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.

Table 4. Example of an Optimisation Rules Repository

Rule_ID	Stat_type	Comp_type	Multi_rows	WHERE_clause	...	Recommendation
101	INSERT	SQL	YES	N/A		INSERT_ALL
102	INSERT	NoSQL/MongoDB	YES	N/A		BULK_INSERT
103	INSERT	NoSQL/Cassandra	YES	N/A		BATCH_INSERT
...						
201	UPDATE	SQL	YES	YES/NO		PARALLEL
202	UPDATE	NoSQL/MongoDB	YES	YES/NO		UPDATE_MANY
203	UPDATE	NoSQL/Cassandra	YES	YES/NO		BATCH_UPDATE
...						
301	DELETE	SQL	YES	NO		TRUNCATE
302	DELETE	NoSQL/MongoDB	YES	NO		DROP
303	DELETE	NoSQL/Cassandra	YES	NO		TRUNCATE
...						

Selected examples from the table will be described. For instance, for the specific *INSERT_ALL* rule to be applied, the precondition of inserting multiple rows must be fulfilled (column *Multi_rows* has a *'YES'* value). 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 and Cassandra component, while, once again, *WHERE_clause* is not applicable.

TRUNCATE, DROP and *TRUNCATE* are respective rewriting rules for the SQL, MongoDB and Cassandra *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 rewriting rules for the *UPDATE* statement of Oracle, MongoDB and Cassandra components of the observed Oracle/MongoDB/Cassandra hybrid are *PARALLEL, UPDATE_MANY,* and *BATCH_UPDATE*, respectively. All three mentioned 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 SRC component determines if it can perform statement rewriting and how. The SRC component reads applicable optimisation rules of the input statement from the Optimisation Rules Repository and forwards them to the ESP component. After receiving the return values from the SRC component, the statement execution flow of the new hybrid with SRC is equivalent to the process in the earlier architecture version. The ESP component sends statements for mapping to the SM com-

ponent. 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, rewriting, optional mapping, statement execution in the hybrid's destination component and user notification.

5. The Use of the Hybrid Database with SRC on Oracle/MongoDB/Cassandra Example

The data model, Figure 3, was developed to demonstrate the usage of the current version of the hybrid SQL/NoSQL database with the new SRC component. The UML Class diagram depicts the created model. The hybrid Oracle/MongoDB/Cassandra database selected for testing implements the shown model.

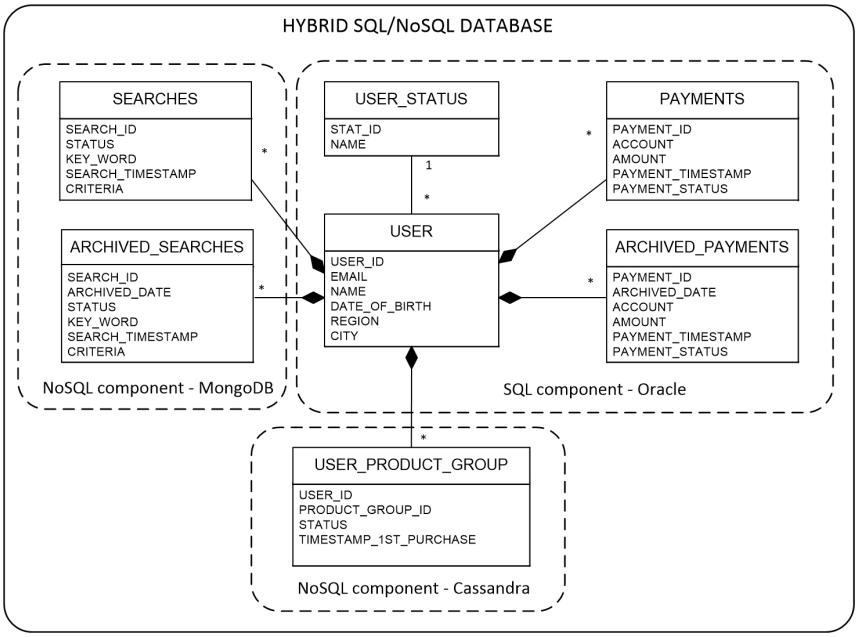

Fig. 3. UML Class diagram for tested domain

The domain of the model is online search and product payment. Figure 3 represents only a part of the model which was needed to realise the use cases chosen for testing (for example, the ordering process was not necessary to show). The model consists of the following classes: *User, User_status, Payments, Searches, Archived_payments,*

Archived_searches, and *User_product_group*, and it is implemented in two versions of the system: the previous hybrid Oracle/MongoDB/Cassandra database without SRC and the current hybrid Oracle/MongoDB/Cassandra database with SRC.

Table 5. Tested use cases for INSERT

Use case id	Statement type and hybrid component	Statement description	Statement before rewriting	Statement after rewriting
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 (MongoDB)	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	INSERT into NoSQL component (Cassandra)	Insert rows into *user_product_group*	INSERT INTO user_product_group (...) VALUES ... INSERT INTO user_product_group (...) VALUES ...	BEGIN BATCH INSERT INTO user_product_group(...) VALUES ... ... INSERT INTO user_product_group(...) VALUES ... APPLY BATCH,

The unconditional consistency of sensitive data, information about users, their statuses, and payments requires storing them 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 (precisely in the MongoDB component). To demonstrate the functioning of the test hybrid SQL/NoSQL database with more than one NoSQL component, an additional Cassandra component was introduced. Cassandra component implements the table *User_product_group*, which contains data of the products group (searched products, bought products, etc.) in correlation with a particular user.

Table 6. Tested use cases for UPDATE

Use case id	Statement type and hybrid component	Statement description	Statement before rewriting	Statement after rewriting
UC_4	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_5	UPDATE NoSQL component (MongoDB)	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_6	UPDATE NoSQL component (Cassandra)	All products' status of the user with *id = 5* updates to value 'Searched'	SELECT distinct product_group_id FROM user_product_group where user_id = 5 ... UPDATE user_product_group SET status = 'Searched' WHERE user_id = 5 and product_group_id=... ... UPDATE user_product_group SET status = 'Searched' WHERE user_id = 5 and product_group_id=...	SELECT distinct product_group_id FROM user_product_group where user_id = 5; BEGIN BATCH UPDATE user_product_group SET status= 'Searched' WHERE user_id = 5 and product_group_id=... ... UPDATE user_product_group SET status= 'Searched' WHERE user_id = 5 and product_group_id = ... APPLY BATCH;

Payments and *Searches* are identifiable and existentially dependent 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 MongoDB 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 realising the mentioned use cases, records are deleted from *Payments* and *Searches*, which represent use cases for deleting all records from the SQL (Oracle) and NoSQL (MongoDB) components, respectively. For the SQL and NoSQL (MongoDB) compo-

nents' update, the 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). Three use cases represent insert, update and delete in the Cassandra component as well. Table 5, Table 6 and Table 7 show the snapshot of use cases chosen for testing based on the hybrid Oracle/MongoDB/Cassandra components.

For every use case, Table 5, Table 6 and Table 7 display 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 rules. The statement rewriting rules, shown in Table 4 and described in Section 4, were applied to nine chosen use cases. All nine selected use cases were executed over the previous version of the hybrid architecture without the SRC component and, after that, over the extended version of the hybrid architecture, which contains the SRC component.

Tests were carried out on the PC with an Intel i7 CPU, with a 2.9 GHz speed, 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 statement 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.

Table 7. Tested use cases for DELETE

Use case id	Statement type and hybrid component	Statement description	Statement before rewriting	Statement after rewriting
UC_7	DELETE from SQL component	Delete all rows from *payments*	DELETE FROM payments	TRUNCATE TABLE payments
UC_8	DELETE from NoSQL component (MongoDB)	Delete all rows from *searches*	db.searches.deleteMany()	db.searches.drop()
UC_9	DELETE from NoSQL component (Cassandra)	Delete all rows from *user_product_group*	SELECT distinct user_id FROM user_product_group... DELETE FROM user_product_group WHERE user_id IN ...	TRUNCATE user_product_group

Measurements of use cases UC_1, UC_2, UC_3, UC_7, UC_8 and UC_9 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_7), NoSQL – MongoDB (UC_2 and UC_8) and NoSQL – Cassandra (UC_3 and UC_9) components were performed over the same amount of records. Use cases UC_4, UC_5 and UC_6 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 statement rewriting rules usage.

6. Experimental Results

The average measured execution times of the use cases chosen for testing are 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 three parts. The first part of the diagram, marked as (a), shows the duration of the statement execution in the SQL (Oracle) hybrid component before and after optimisation, more precisely, statement rewriting. The second 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 statement rewriting, while the third part, labelled as (c), represents the average execution time in the second NoSQL component (Cassandra), also before and after statement rewriting. Before applying statement rewriting, the use case is executed in a hybrid database without SRC (previous architecture), and the after statement rewriting presents the statement execution in a hybrid database with SRC (extended architecture). Although we acknowledge that the statement rewriting represents one of many optimisation techniques, for the easiness of presenting and analysing the following results, by "before/after optimisation", we will mean "before/after applying particular statement rewriting rule".

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. In the Cassandra component, the values before applying the optimisation recommendation were 5.75 (5,000 records), 7.729 (10,000 records), 14.815 (30,000 records), 22.281 (50,000 records), 30.937 (75,000 records) and 39.995 (100,000 records) seconds. After SRC executed statement rewriting, the average execution times were significantly decreased to 0.869 (5,000 records), 1.257 (10,000 records), 1.746 (30,000 records), 2.345 (50,000 records), 2.981 (75,000 records) and 3.374 (100,000 records) seconds.

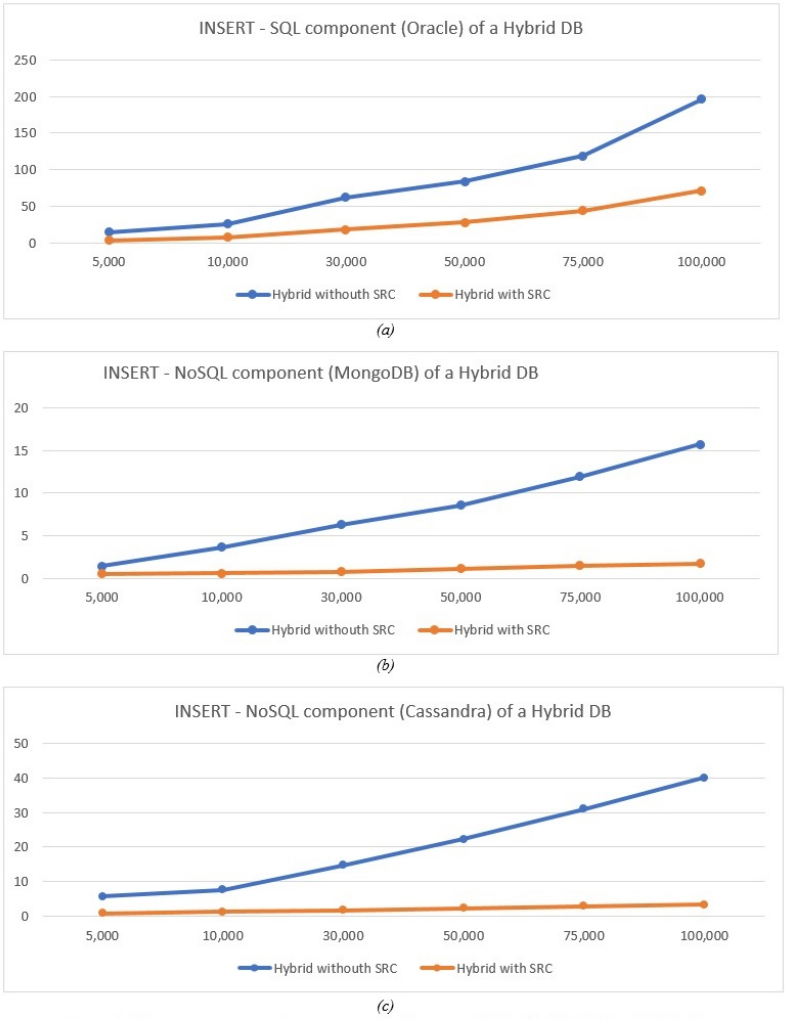

Fig. 4. The average measured execution times of use cases UC_1 (a), UC_2 (b) and UC_3 (c)

The optimised (rewritten) INSERT statement uses INSERT ALL and BULK INSERT for the SQL and NoSQL components, respectively, and achieves shorter execution times over a hybrid without SRC, 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. Cassandra is representative of the wide-column NoSQL databases. As shown by the achieved results, concerning the schema structure strictness and consistency, Cassandra is between MongoDB and Oracle but closer to Oracle. That can be concluded by achieving a noticeably higher aver-

age execution time than MongoDB, especially for the non-optimised statements. On the other hand, Cassandra still manages shorter execution times than Oracle, benefiting from the wide-column principle of storing data. Cassandra achieved a shorter pre-optimised execution time (on 100,000 records than Oracle on 30,000), which became even more emphasized with the rewritten statements (quicker on 100,000 records than Oracle on 10,000).

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 inserted 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. Because of that, the mentioned fixed cost of statement concatenation was multiplied. However, the average execution time over a larger amount of data highlighted the benefits of executing one INSERT ALL over 1,000 records.

Figure 5 shows the average UC_4, UC_5 and UC_6 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 SRC, 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_4 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_5, on the hybrid without SRC, 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_5 achieved drastically decreased average execution times. The average times for UC_5 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_5. 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.

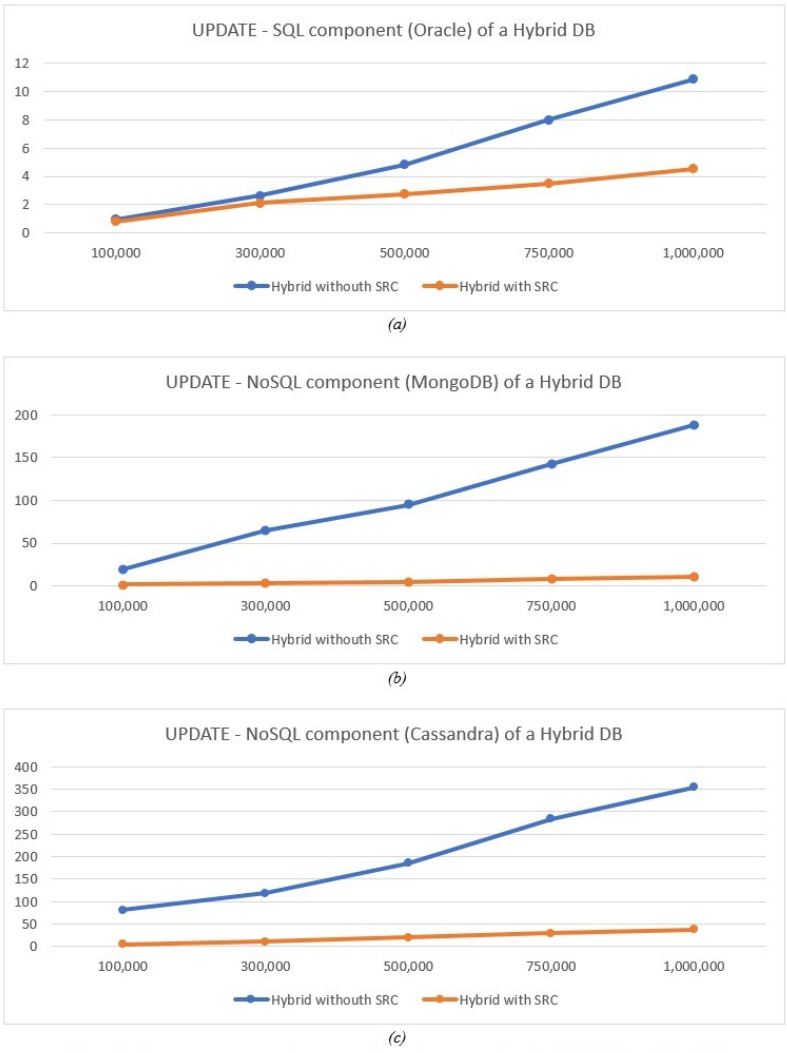

Fig. 5. The average measured execution times of use cases UC_4 (a), UC_5 (b) and UC_6 (c)

After the optimisation, a significant decrease in average execution time also occurred within the Cassandra component. UPDATE statement in the Cassandra component, inside the architecture without SRC, achieved 80.766 (100,000 records), 118.905 (300,000 records), 185.432 (500,000 records), 284.124 (750,000 records) and 354.174 (1,000,000 records) seconds. In comparison, the rewritten statement in Cassandra inside the new architecture with SRC achieved 4.343 (100,000 records), 11.172 (300,000 records), 19.781 (500,000 records), 29.7 (750,000 records) and 37.211 (1,000,000 records) seconds. Because in the tested version of Cassandra and its driver, BATCH UPDATE was successfully

executed with no more than 300,000, four calls of this syntax (for 1,000,000 records) were executed. Still, they were noticeably quicker than the non-optimised multiple UPDATE, which has an obligatory WHERE clause with all primary key fields and without the support of IN.

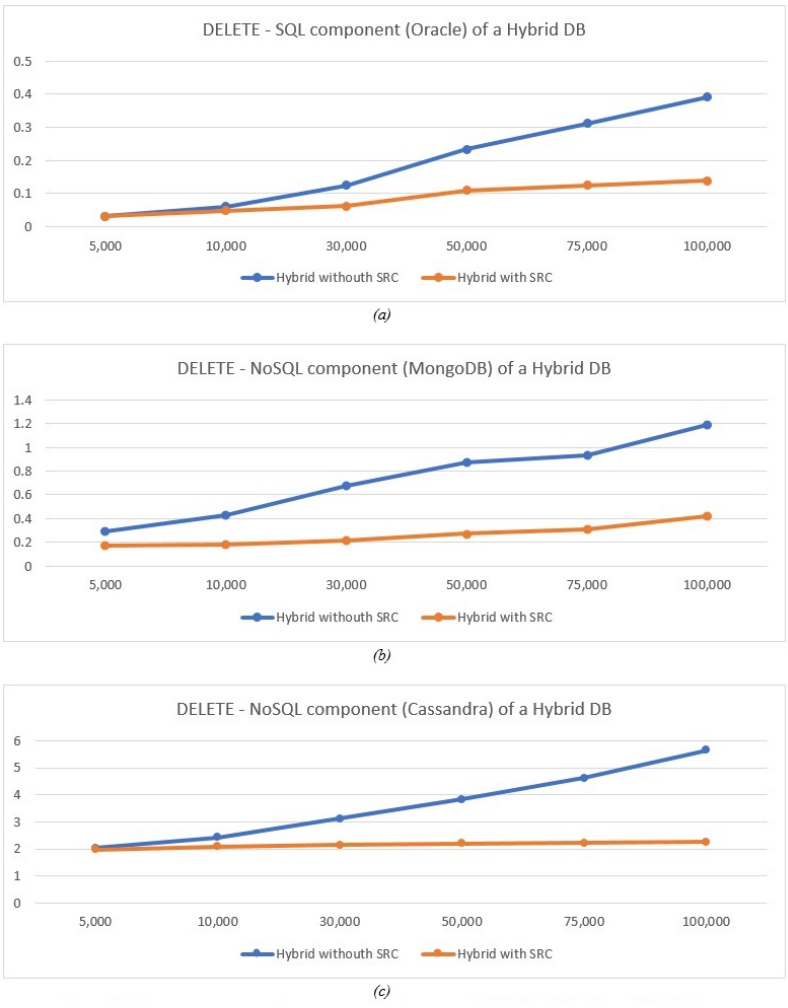

Fig. 6. The average measured execution times of use cases UC_7 (a), UC_8 (b) and UC_9 (c)

Figure 6 depicts the optimisation effects of UC_7, UC_8 and UC_9 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 has achieved 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_8 performance before optimisation. The optimised UC_8 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.

Using the TRUNCATE statement in the Cassandra component decreased the average execution time in the architecture with the SRC. Old architecture without SRC achieved average of 2.032 (5,000 records), 2.433 (10,000 records), 3.109 (30,000 records), 3.823 (50,000 records), 4.616 (75,000 records) and 5.656 (100,000 records) seconds, while the new one, with SRC, achieved 1.965 (5,000 records), 2.081 (10,000 records), 2.137 (30,000 records), 2.191 (50,000 records), 2.225 (75,000 records) and 2.25 (100,000 records) seconds. Although the recommendation for the DELETE statement wasn't as dominant as the BATCH technique for INSERT or DELETE, it still managed time decrease. It is noticeable that with the smaller datasets (for example, 5,000 records), there is nothing to separate DELETE FROM and TRUNCATE. Still, with the increased dataset volume, a slight advantage is on the rewritten statement side.

7. Conclusions and Future Work

The findings presented in this paper represent the continuation of the hybrid SQL/NoSQL databases research. The motivation was to give answers to the research questions which emerged during the previous phase of research. The main goal was to explore the feasibility and justification of extending the hybrid SQL/NoSQL database by creating new Statement Rewriting Component and Optimisation Rules Repository components and integrating them into the well-proven hybrid's architecture. As support for applying rewriting techniques, a process model (in the UML notation) was developed.

Without striving to cover all possible optimisation rules for all types of databases, which would be an almost impossible task at the moment, selected statement rewriting rules were chosen for INSERT, UPDATE and DELETE statements. Test use cases demonstrate the average statement duration over the hybrid database with and without SRC. In some use cases, over certain records, a hybrid database with the SRC component didn't necessarily achieve a shorter execution time (i.e. UC_7 over 5,000 records). However, the observed trend was that the hybrid database with the SRC component required less time for execution when the number of records was increased compared to the hybrid without SRC.

The conclusion that arises is that with the smaller number of records, a hybrid with SRC cannot always necessarily achieve a decrease in the average execution time in comparison to the hybrid without SRC. However, when working with a larger amount of data (several tens or hundreds of thousands of records), experimental tests indicated a significant decrease in the average execution times for the selected use cases of the presented domain. These results were gathered on a particular Oracle/MongoDB/Cassandra hybrid, which was chosen for simulating execution in the architecture with and without the SRC component. However, we acknowledge that these results present one instance of outcomes and that the architecture extended with the SRC is still in the prototype phase. Nevertheless, the results showed the purposes of the introduced extension and the expected performance-gaining trend of using a hybrid with SRC.

There are several identified directions of future work to overcome the limitations of the current version of hybrid SQL/NoSQL databases. The first one is to implement additional functionalities into the current prototype architecture. That will enable expansion of for-now supported basic optimisation techniques for SELECT and their combining with other DML statements (i.e. SELECT with complex subquery, but also UPDATE and DELETE with subquery). The current version doesn't support DDL statements, and incorporating these functionalities would enable users to easily create and alter all types of database objects, no matter what component of a hybrid would store it, instead of just manipulating with data in the existing objects. An important direction of future research and advancing the presented approach would be expanding the number of different components inside the hybrid while introducing particular implementations for the other subtypes of databases (for example, key-value and graph) and the syntax support for other SQL DBMS (for example, PostgreSQL, MS SQL Server etc.) as well as other NoSQL systems. Additional enhancement of the present architecture could include broadening optimisation techniques in the Optimisation Rules Repository because the number of statement optimisation rules is not final and can be extended. In the end, expanding testing Oracle/MongoDB/Cassandra hybrid database to a more complex system with multiple components of many other types of databases is planned for the future.

References

1. C. A. Lana, M. Guessi, P. O. Antonino, D. Rombach, and E. Y. Nakagawa. A systematic identification of formal and semi-formal languages and techniques for software-intensive systems-of-systems requirements modeling. *IEEE Systems Journal*, 13(3):2201–2212, 2019.
2. V. de Oliveira Neves, A. Bertolino, G. De Angelis, and L. Garcés. Do we need new strategies for testing systems-of-systems? In *Proceedings of the SESoS'18: SESoS'18:IEEE/ACM 6th International Workshop on Software Engineering for Systems-of-Systems*, pages 29–32, New York, NY, USA, 2018. ACM.
3. A. Bertolino and R. Mirandola. Software performance engineering of component-based systems. In *Proceedings of the Fourth International Workshop on Software and Performance, WOSP 2004*, pages 238–24, Redwood Shores, California, USA, 2004. Association for Computing Machinery, NY, United States.
4. A. Bertolino, G. De Angelis, and F. Lonetti. Governing regression testing in systems of systems. In *Proceedings of 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 144–148, Berlin, Germany, 2019. IEEE.

5. S. Park, Y. Shin, S. Hyun, and D. Bae. Simva-sos: Simulation-based verification and analysis for system-of-systems. In *Proceedings of the 15th International Conference of System of Systems Engineering (SoSE)*, pages 575–580, Budapest, Hungary, 2020. IEEE.
6. M.A. Olivero, A. Bertolino, F.J. Dominguez-Mayo, M.J. Escalona, and I. Matteucci. Addressing security properties in systems of systems: Challenges and ideas. In R. Calinescu and F. Di Giandomenico, editors, *Software Engineering for Resilient Systems - SERENE 2019*, volume 11732 of *Lecture Notes in Computer Science*, pages 138–146. Springer, Cham, 2019.
7. H. Cadavid, V. Andrikopoulos, and P. Avgeriou. Improving hardware/software interface management in systems of systems through documentation as code. *Empirical Software Engineering*, 28, 2023.
8. ANSI. Ansi/x3 /sparc dbms framework. Report of the Study Group on Database Management Systems, 1977.
9. B. Lazarević, Z. Marjanović, N. Aničić, and S. Babarogić. *Baze podataka*. FON, Belgrade, Serbia, 2006.
10. A. Borgida, M. Casanova, and A. H. F. Laender. Logical database design: from conceptual to logical schema. In L. LIU and T. ÖZSU, editors, *Encyclopedia of Database Systems*, pages 1645–1649. Springer, Boston, MA, US, 2009.
11. S. Bjeladinovic. A fresh approach for hybrid sql/nosql database design based on data structuredness. *Enterprise Information Systems*, 12(8-9):1202–1220, 2018.
12. S. Bjeladinovic, Z. Marjanovic, and S. Babarogic. A proposal of architecture for integration and uniform use of hybrid sql/nosql database components. *Journal of Systems and Software*, 168:110633, 2020.
13. H.R. Vyawahare, P.P. Karde, and V.M. Thakare. A hybrid database approach using graph and relational database. In *Proceedings of the 2018 IEEE International Conference on Research in Intelligent and Computing in Engineering*, pages 2555—-2564, Univ Don Bosco, San Salvador, EL SALVADOR, 2018. IEEE.
14. SolidIT. Db-engines ranking. Web site: DB-engines ranking, 2024. [Online]. Available on: https://db-engines.com/en/ranking (Retrieved: January 2024).
15. K. Sudhakar. Difference between sql and nosql databases. *International Journal of Management, IT and Engineering*, 8(6):444–452, 2018.
16. A. Faraj, B. Rashid, and T. Shareef. Comparative study of relational and nonrelations database performances using oracle and mongodb systems. *International Journal of Computer Engineering Technology (IJCET)*, 5(11):11–22, 2014.
17. A. Vágner. How do nosql databases handle variety of big data? In XS. Yang, S. Sherratt, and Joshi A. Dey, N., editors, *Proceedings of Ninth International Congress on Information and Communication Technology ICICT 2024*, volume 1012 of *Lecture Notes in Networks and Systems*, pages 459–469. Springer, Singapore, 2024.
18. L. Zhang, K. Pang, J. Xu, and B. Niu. Json-based control model for sql and nosql data conversion in hybrid cloud database. *Journal of Cloud Computing*, 11(23), 2022.
19. S. Goyal, P.P. Srivastava, and A. Kumar. An overview of hybrid databases. In *Proceedings of the 2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, pages 285–288, Greater Noida, India, 2015.
20. C. Gyorodi, R. Gyorodi, and R. Sotoc. 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, 2015.
21. B. James and P.O. Asagba. Hybrid database system for big data storage and management. *International Journal of Computer Science, Engineering and Applications (IJCSEA)*, 7(3/4):15–27, 2017.
22. N. Jatana, S. Puri, M. Ahuja, I. Kathuria, and D Gosain. A survey and comparison of relational and non-relational database. *International Journal of Engineering Research Technology*, 1(6):1–5, 2012.

23. M. Villari, A. Celesti, M. Giacobbe, and M. Fazio. Enriched e-r model to design hybrid database for big data solutions. In *Proceedings of the 2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 163–166, Messina, Italy, 2016. IEEE.
24. D. Martinez-Mosquera, R. Navarrete, and S. Lujan-Mora. Modeling and management big data in databases—a systematic literature review. *Sustainability*, 12(2):634, 2020.
25. J. Duggan, A. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. The bigdawg polystore system. *ACM SIGMOD Record*, 44(2):11–16, 2015.
26. E. Kharlamov, T. Mailis, K. Bereta, D. Bilidas, S. Brandt, E. Jimenez-Ruiz, S. Lamparter, C. Neuenstadt, O. Özçep, A. Soylu, C. Svingos, G. Xiao, D. Zheleznyakov, D. Calvanese, I. Horrocks, M. Giese, Y. Ioannidis, Y. Kotidis, R. Moller, and A. Waaler. A semantic approach to polystores. In *Proceedings of the 2016 IEEE International Conference on Big Data*, pages 2565–2573, Washington, DC, USA, 2016. IEEE.
27. S. Dasgupta, K. Coakley, and A. Gupta. Analytics-driven data ingestion and derivation in the awesome polystore. In *Proceedinsg of the 2016 IEEE International Conference on Big Data*, pages 2555–2564, Washington, DC, USA, 2016. IEEE.
28. A. Maccioni, E. Basili, and R. Torlone. Quepa: Querying and exploring a polystore by augmentation. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2133–2136, San Francisco, California, USA, 2016. Sigmod.
29. J. McHugh, P.E. Cuddihy, J.W. Williams, K.S. Aggour, V.S. Kumar, and V. Mulwad. Integrated access to big data polystores through a knowledge-driven framework. In *Proceedings of the 2017 IEEE International Conference on Big Data*, pages 1494–1503, Boston, MA, USA, 2017. IEEE.
30. F. Basciani, J. Di Rocco, L. Iovino, and A. Pierantonio. Typhonml: Tool support for hybrid polystor. *Science of Computer Programming*, 232:103044, 2023.
31. N. Niu, L. D. Xu, and Z. Bi. Enterprise information systems architecture - analysis and evaluation. *IEEE Transactions On Industrial Informatics*, 9(4):2147–2154, 2013.
32. O. Lajam and S. Mohammed. Revisiting polyglot persistence: From principles to practice. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 13(5):872–882, 2022.
33. E. Płuciennik and K. Zgorzałek. The multi-model databases – a review. In S. Kozielski, D. Mrozek, P. Kasprowski, B. Małysiak-Mrozek, and D. Kostrzewa, editors, *Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation. BDAS 2017.*, volume 716 of *Communications in Computer and Information Science*, pages 141–152. Springer, Cham, 2017.
34. J. Lu and I. Holubová. Multi-model databases. *ACM Computing Surveys*, 52(3):1–38, 2019.
35. J. Lu, I. Holubová, and B. Cautis. Multi-model databases and tightly integrated polystores. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 2301–2302, New York, NY, USA, 2018. Association for Computing Machinery.
36. F. Ye, X. Sheng, N. Nedjah, J. Sun, and P. Zhang. A benchmark for performance evaluation of a multi-model database vs. polyglot persistence. *Journal of Database Management*, 34(3):1–20, 2023.
37. D. Van Landuyt, J. Benaouda, V. Reniers, A. Rafique, and W. Joosen. A comparative performance evaluation of multi-model nosql databases and polyglot persistence. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 286—-293, New York, NY, USA, 2023. Association for Computing Machinery.
38. I. Holubová, M. Vavrek, and S. Scherzinger. Evolution management in multi-model databases. *Data Knowledge Engineering*, 136:101932, 2021.
39. Van Landuyt D. Rafique A. Joosen W. Reniers, V. Object to nosql database mappers (ondm): A systematic survey and comparison of frameworks. *Information Systems*, 85:1–20, 2019.

40. N. Roy-Hubara and A. Sturm. Design methods for the new database era: a systematic literature review. *Software and Systems Modeling*, 19:297–312, 2020.
41. A. Kalayda. Promising directions for the development of modern databases. *Journal of Physics: Conference Series*, 2131(022087):1–6, 2021.
42. B. Bender, C. Bertheau, T. Kŏrppen, H. Lauppe, and N Gronau. A proposal for future data organisation in enterprise systems—an analysis of established database approaches. *Information Systems and e-Business Management*, 20:441—494, 2022.
43. I. Zečević, P. Bjeljac, B. Perišić, S. Stankovski, D. Venus, and G. Ostojić. Model driven development of hybrid databases using lightweight metamodel extensions. *Enterprise Information Systems*, 12(8-9):1221–1238, 2018.
44. H.N. Aleem, M.M. Baig, and M.M. Khan. Efficient software testing technique based on hybrid database approach. *International Journal of Advanced Computer Science and Applications*, 10(7):349—-356, 2019.
45. H.R. Vyawahare, P.P. Karde, and V.M. Thakare. Hybrid database model for efficient performance. *Procedia Computer Science*, 152(8-9):172–178, 2019.
46. A. de la Vega, D. García-Saiz, C. Blanco, M. Zorrilla, and P. Sánchez. Mortadelo: A model-driven framework for nosql database design. In E. Abdelwahed, L. Bellatreche, M. Golfarelli, D. Méry, and C. Ordonez, editors, *Model and Data Engineering (MEDI 2018)*, volume 11163 of *Lecture Notes in Computer Science*, pages 41–57. Springer, Cham, 2018.
47. M. Sokolova, F. Gómezb, and L. Borisoglebskayaa. Migration from an sql to a hybrid sql/nosql data model. *Journal of Management Analytics*, 7(1):1–11, 2019.
48. F. Abdelhedi, A.A. Brahim, F. Atigui, and G. Zurfluh. Logical unified modeling for nosql databases. In *Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS 2017)*, pages 249–256, Porto, Portugal, 2017. HAL Science.
49. K. Mershad and A. Hamieh. Sdms: smart database management system for accessing heterogeneous databases. *International Journal of Intelligent Information and Database Systems*, 14(2):115–152, 2021.
50. L. Nikolic, V. Dimitrieski, and M. Celikovic. An approach for supporting transparent acid transactions over heterogeneous data stores in microservice architectures. *Computer Science and Information Systems*, 21(1):167—-202, 2024.
51. O. Mehdi, H. Ibrahim, S. Affendey, E. Pardede, and J. Cao. Exploring instances for matching heterogeneous database schemas utilizing google similarity and regular expression. *Computer Science and Information Systems*, 15(2):295–320, 2018.
52. R. Čerešňák and M. Kvet. Comparison of query performance in relational a non-relation databases. *Transportation Research Procedia*, 40:170–177, 2019.
53. K. Fraczek and M. Plechawska-Wojcik. Comparative analysis of relational and non-relational databases in the context of performance in web applications. In S. Kozielski, D. Mrozek, P. Kasprowski, B. Małysiak-Mrozek, and D. Kostrzewa, editors, *Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation. BDAS 2017.*, volume 716 of *Communications in Computer and Information Science*, pages 153–164. Springer, Cham, 2017.
54. Z.H. Liu, B. Hammerschmidt, D. McMahon, Y. Liu, and H.J. Chang. Closing the functional and performance gap between sql and nosql. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*, pages 227–238, San Francisco, USA, 2016. Sigmod.
55. S. Bjeladinović, M. Škembarević, O. Jejić, and M. Asanović. An analysis of using binary json versus native json on the example of oracle dbms. *IPSI Transactions on Internet Research*, 19(2):92–103, 2023.
56. A. Kemper and T. Neumann. One size fits all, again! the architecture of the hybrid oltpolap database management system hyper. In M. Castellanos, U. Dayal, and V. Markl, editors, *Enabling Real-Time Business Intelligence (BIRTE 2010)*, volume 84 of *Lecture Notes in Business Information Processing*, pages 7–23. Springer, Berlin, Heidelberg, 2011.

57. L. Thiry, H. Zhao, and M. Hassenforder. Categories for (big) data models and optimisation. *Journal of Big Data*, 5(21), 2018.
58. B. Scheuermann. Design of a reconfigurable hybrid database system. In *Proceedings of the 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 247–250, Charlotte, NC, USA, 2010. IEEE.
59. M. Owaida, D. Sidler, K. Kara, and G. Alonso. Centaur: A framework for hybrid cpu-fpga databases. In *Proceedings of the 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2017)*, pages 211–218, Napa, CA, USA, 2017. IEEE.
60. S. Breß, E. Schallehn, and I. Geist. Towards optimization of hybrid CPU/GPU query plans in database systems. In M. Pechenizkiy and M. Wojciechowski, editors, *Advances in Intelligent Systems and Computing*, volume 185 of *Advances in intelligent systems and computing*, pages 27–35. Springer Berlin Heidelberg, 2013.
61. S. Cremer, M. Bagein, S. Mahmoudi, and P. Manneback. Improving performances of an embedded relational database management system with a hybrid cpu/gpu processing engine. In C. Francalanci and M. Helfert, editors, *Data Management Technologies and Applications. DATA 2016*, volume 737 of *Communications in Computer and Information Science*, pages 160–177. Springer, Cham, 2017.
62. M. Gowanlock, B. Karsin, Z. Fink, and J. Wright. 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)*, pages 1–11, Amsterdam Netherlands, 2019. ACM.
63. Z. Pang, S. Wu, H. Huang, Z. Hong, and Y. Xie. Aqua+: Query optimisation for hybrid database-mapreduce system. *Knowledge and Information Systems*, 63:905—938, 2021.
64. W. Khan, W. Ahmad, B. Luo, and E. Ahmed. Sql database with physical database tuning technique and nosql graph database comparisons. In *Proceedings of the 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC 2019)*, pages 110–116, Chengdu, China, 2019. IEEE.
65. R. Sellami and B. Defude. Complex queries optimisation and evaluation over relational and nosql data stores in cloud environments. *IEEE Transactions on Big Data*, 4(2):217–230, 2018.
66. C. Li and J. Gu. An integration approach of hybrid databases based on sql in cloud computing environment. *Software: Practice and Experience*, 49(3):11–16, 2018.
67. SolidIT. Db-engines ranking – relational dbms. Web site: DB-engines ranking, 2024. [Online]. Available on: https://db-engines.com/en/ranking/relational+dbms (Retrieved: January 2024).
68. SolidIT. Db-engines ranking – document store dbms. Web site: DB-engines ranking, 2024. [Online]. Available on: https://db-engines.com/en/ranking/document+store (Retrieved: January 2024).
69. SolidIT. Db-engines ranking – wide-column store dbms. Web site: DB-engines ranking, 2024. [Online]. Available on: https://db-engines.com/en/article/Wide+Column+Stores (Retrieved: December 2024).
70. C. Nance, T. Losser, R. Iype, and G. Harmon. Nosql vs rdbms - why there is room for both. In *Proceedings of the Southern Association for Information Systems Conference*, pages 111–116, 2013.

Srđa Bjeladinović is an Assistant Professor at Faculty of Organizational Sciences, University of Belgrade. He received his M.Sc. and Ph.D. degrees in Information Systems from University of Belgrade. His research interests are databases, information systems development methodologies, integrated software solutions and ERPs. In recent years he has been researching NoSQL and hybrid databases.

Received: October 24, 2024; Accepted: February 20, 2025.