The Journal of Systems & Software 168 (2020) 110633

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A proposal of architecture for integration and uniform use of hybrid SQL/NoSQL database components

Srdja Bjeladinovic *, Zoran Marjanovic, Sladjan Babarogic

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

ARTICLE INFO

Article history:
Received 4 March 2019
Received in revised form 21 January 2020
Accepted 5 May 2020
Available online 16 May 2020

Keywords:
Architecture proposal
Database integration
Uniform use
Hybrid database
SQL
NoSQL

ABSTRACT

The popularity of social networks and the expansion of various second-generation Internet services have contributed to the increase of data, of different structuredness levels, in use. Relational databases (frequently called SQL databases) pose themselves as a logical choice for the management of data containing fixed or rarely changeable structure. The need for fast processing of vast quantities of unstructured data has opened the door for the rise of NoSQL databases popularity. The business of modern organizations often faces the challenge of parallel use of different database types. In recent years, hybrid SQL/NoSQL databases, which contain SQL and NoSQL databases as its components, become a popular solution for the issue above. This paper identifies and describes a possible way of integration and uniform use (as two significant non-functional requirements) of hybrid database components, as well as introduce the architecture for this purpose. The presented architecture, with its specially developed components, provides as simple usage as a single database does, with advantages of parallel use of databases of different types. The functioning principle of the new architecture is elaborated on a series of practical use cases of various complexities, which were tested against a hybrid database, and Oracle and MongoDB as well.

1. Introduction

The present-day sees the use of various types of databases (SolidIT, 2019). Among the most popular are still relational databases, also called SQL databases (named after the query language they support). Since introduce, SQL databases have been providing the users with a high level of efficiency for storage and management of structured data (the data of precisely defined structure, which is non-frequently changeable) (Arenas et al., 2014; Shepelev, 2011; Lebo et al., 2017; Nickel and Tresp, 2013). That efficiency is especially noticeable when providing data consistency and executing the complex queries (with a high number of join operations) are set as important requirements. The increase of the number of social media users and further expansion of Internet services (Abiteboul et al., 2011), which have influenced the business by actualizing the need for fast processing of the vast amount of not always structured data, has contributed to NoSQL databases' popularity, as well as their sub-types' popularity (key–value, column-based, document-based and graph databases) (Sadalage and Fowler, 2012). NoSQL databases support flexible and easily changeable schemas (or the lack of the same) (Lake and Crowther, 2013; Sullivan, 2015). Besides that, they prioritize the data to be available at the expense of their consistency thus allowing efficient management of unstructured data or the data of low structuredness level (Inmon et al., 2008).

The modern organizations' functioning frequently deals with a need to reconcile opposing requirements, provided by databases of different types. Thus, for example, it is necessary to enable parallel use of data of varying level of structuredness regardless of the source of the data (e.g., whether they are structured data regarding bank transactions or unstructured data from social media) as well as to simultaneously guarantee data integrity with maximum availability within one business system. The popularity and specific characteristics of SQL and NoSQL databases have helped justify their parallel use in modern businesses due to their ability to retain the efficiency while being used for their respective purposes (Avison and Fitzgerald, 2003; Badia and Lemire, 2011; da Silva, 2011; Nance et al., 2013; Nyati et al., 2014). In support of this, the additional argument is that leading database manufacturers, such as Oracle and Microsoft, have in the palette of their products both SQL and NoSQL DBMS and are working on their integration (Microsoft, 2019; Oracle, 2019a). For the mentioned reasons, it is not always optimal for organizations to decide on only one of the database types. Instead, it is necessary to provide the organizations with an opportunity of parallel use of SQL and NoSQL databases, intending to profit from the best functionalities of both worlds.

* Corresponding author.
E-mail addresses: srdja.bjeladinovic@fon.bg.ac.rs (S. Bjeladinovic), zoran.marjanovic@fon.bg.ac.rs (Z. Marjanovic), sladjan.babarogic@fon.bg.ac.rs (S. Babarogic).

https://doi.org/10.1016/j.jss.2020.110633
0164-1212/

In the process of design, each system has to meet functional, user and business requirements. Since a system that does not satisfy them cannot be expedient, nor can it be charged, these categories of requirements are considered mandatory. However, there is a wide range of qualitative attributes in modern enterprise information systems that fall into the category of Non-Functional Requirements (NFRs). No system can fulfil all NFRs, as some of them are opposed (e.g. performance versus resource usage or security versus flexibility). However, before designing the system, it is necessary to determine desirable NFRs. A detailed representation of the papers dealing with NFRs was provided by authors (Niu et al., 2013). They organized all NFRs according to the criterion of key qualitative attributes. They listed more specific terms related to each NFR, which are semantically close to those key qualitative attributes (e.g. terms Interoperability, Scalability and Coordination are close to the key attribute of Integration, and Simplicity is close to the Usability). All NFRs are ranked on a scale from those that are business-driven to those that are dominantly software-driven.

By analysing classified requirements (Niu et al., 2013), the following NFRs were identified as particularly significant for addressing the need for parallel use of different type databases:

(1) Integration — one of the leading business-driven NFRs
(2) Usability (uniform use) — one of the leading software-driven NFRs.

The integration of SQL and NoSQL databases must be fulfilled in order to provide a system with a single logical database, rather than multiple separate databases (which are difficult to administer and make it almost impossible to comprehensively manage the data model, structures in use, integrity rules, as well as redundancy). Usability, on the other hand, guarantees that a complex system, using databases of different types, will maintain an acceptable level of simplicity. In order to maintain simplicity when developing and maintaining a system containing databases of different types, all its components should be used uniformly. Uniform use is one of the main indicators of simplicity, and therefore the system usability.

Over time, different approaches have emerged to address the integration and uniform usage of SQL and NoSQL databases. Hybrid databases stood out among them. Unlike other approaches, such as polystores (Duggan et al., 2015), which have multiple data models and inevitably contain redundancy, hybrids provide integration through a single data model, with support for different data structures used for its implementation, i.e. for storing data. This establishes integration between the various databases which represent hybrid components, while usability is achieved through the usage of a single language for managing all the data of all components. Existing solutions have opened up space for further increasing of the integration level, which can be reflected through the process of unified administration of the entire logical database. Centralized administration can provide comprehensive management of the data model, all structures, rules, constraints, and redundancy, enabling tighter integration of different databases. This represents a major improvement introduced by hybrids. In order to provide an adequate level of integration of the hybrid components and to provide the user with a sense of usability and simplicity in working with the hybrid database, it was necessary to develop a dedicated architecture, with specific extensions. The development of architecture was preceded by the profiling of the following research questions

(1) What processes should be supported and what components should the new architecture contain to enable the integration and uniform use (two initial NFRs) of SQL and NoSQL databases as components of a hybrid database?
(2) What novelties does the new architecture bring over existing architectures, and how can they be used to solve the integration challenges of other databases?
(3) How does the new architecture affect performance?

The aim of this paper (stands out from all of the above) is to propose an architecture suitable for providing integration and uniform use of SQL and NoSQL databases as components of a hybrid SQL/NoSQL database. The approach presented in this paper identifies specific processes significant for integration and uniform use of hybrid's components and presents dedicated architecture design, which represents a widening of three-tier architecture with the addition of specially developed new components.

The suggested architecture has been developed as a natural support to the hybrid SQL/NoSQL database design approach (Bjeladinovic, 2018) even though it may, with potential adjustments, generally be applied for other database integration approaches.

The overview of different approaches for integration and uniform usage of SQL and NoSQL databases is presented in Section 2. Section 3 contains the specification of the new architecture with specially developed architectural components for integration and uniform use of hybrid database components. Section 4 shows selected business use cases. Process of the statement execution in the new architecture is explained, from the moment the user enters the statement to the moment of completion of a use case. Section 5 contains the results of executed tested on selected use cases. Test have been conducted on hybrid SQL/NoSQL database, and on SQL and NoSQL databases as well. Key analysis and directions of further development are displayed at the end.

2. Related work

The need for integration and uniform use of SQL and NoSQL databases stems from their differences. The two main distinguishing aspects of SQL and NoSQL databases are:

(1) Data models and the corresponding structures they use to implement them
(2) Query languages and supported operations

One of the essential differences between SQL and NoSQL databases represent the data model they support. The relational model, on which are based SQL databases, allows a formal definition of model structure, operations, and integrity rules. They use standardized SQL query for operations on data. Integrity rules consist of entity rules and referential integrity rules. The structure of a model is defined using relations, domains, keys, and database schema, while a table is a typical structure for storing data. NoSQL databases can be from one of four subcategories, which are named based on the data model they support: key–value, column family, document and graph. Although there have been attempts to define a common model for the design of all NoSQL databases (Bugiotti et al., 2014), still the characteristics of the concrete data model, which uses particular NoSQL representative, dictate the specifics in the design process. Document and column-family databases base their model on the extension of the key–value model, where the value of the corresponding key is the unique identifier of the document or the column family. Similarly, graph nodes also have identifiers. Accordingly, it can be considered that the mentioned types of databases support entity rules, while such observation cannot be claimed for referential integrity. Because NoSQL databases are schema-less, the absence of a strictly defined schema structure eliminates the possibility of tightly defining and checking the referential integrity rules, which results in a lower degree of control over data values. Redundancy and de-normalization represent a desirable property and a frequently used technique in NoSQL databases design, as

opposed to SQL databases that strive for strong control, even elimination, of redundancy. The drawback of NoSQL databases is the absence of a standardized language for working with data, as a consequence of that query languages have been defined separately for each manufacturer of DBMS. Comparison of commonly used operations and clauses between SQL and several NoSQL languages are given in Table 1.

Table 1 shows the absence of JOIN operation support within the representatives of NoSQL databases, which can be explained by the fact that NoSQL does not fully support the rules of referential integrity and that de-normalization is used as a frequent design technique. NoSQL databases compensate the lack of JOIN operation either by operators for referencing values (e.g. lookup) or by nesting data (e.g. nested documents). However, operators such as lookup can replace the JOIN clause only in particular sense (due to the complexity of usage and the difficulty of accessing individual, non-aggregated data Buckler, 2016). Also, the absence of strict referential integrity constraints requires additional application coding in order to verify the value of the data, as otherwise, it may violate the integrity of the data. A more detailed analysis of the differences between querying services and the operations they support has been addressed by the authors (van Ombergen, 2014; Grolinger et al., 2013; Bach and Werner, 2014; Han et al., 2011; Holzschuher and Peinl, 2013).

In this paper, it is particularly important how, already listed, differences in data models, storage structures, integrity rules, and differences in query languages (and especially the absence of the ability to directly join data from different types of databases) affect the architecture that should reconcile these differences, enable the integration of heterogeneous databases and integrate them into a single logical database. These challenges have been analysed in existing papers, which have proposed different architectural specifications. The analysed works have opened the space for further integration and usability of different type databases. Also, they have enabled the emergence of a new extension of the three-tier architecture, with its newly developed components, presented in this paper. Depending on the manner used for expanding three-tier architecture, available papers are possible to be divided into the following categories:

- Approaches which adjust and expand the presentational layer as a point of unification access to databases of different types,
- Approaches which expand the architecture with a component or a layer in charge of language mapping between databases of different types (SQL and NoSQL).

A common characteristic of approaches from the first category is that with the development of dedicated interfaces for data-intensive systems of various structuredness levels, the entry point for filling database is unified, regardless of the database type (SQL or NoSQL). Thus, some authors (Zhu and Wang, 2012) state that designing a common user interface may play the role of an integrator of different parts of the system, which uses Big Data. Other authors (Polese and Vacca, 2009) suggest introducing Interface manager, an extension of the presentational layer, which would enable an alteration of user query with the aim of adjustment of the entered query with the current database version. In their work, authors (Atzeni et al., 2012), using meta-model design approach, created a generic interface called SOS (Save Our Systems) which enables concrete interfaces to be mapped into generic ones with abstracting of their specificity. This was demonstrated on an example of interface integration among MongoDB, Redis and HBase.

The second approach category contains the works which suggest the introduction of component or layer for query mapping of different database types. One possible solution, implemented through SQL queries mapping into HBase queries, is presented by authors (Vilaça et al., 2013). Others (Tatemura et al., 2012) offered a more comprehensive suggestion to create a SQL mapper, which would enable the use of SQL query over NoSQL databases. The above-mentioned work presents a layer called Partiqle, used for SQL query mapping into query suitable for NoSQL databases. They demonstrated the usage of Partiqle on HBase. These papers have attributed to some extent in creating a solution regarding SQL and NoSQL databases integration. Despite the general idea of using SQL language with NoSQL systems, the downside of these approaches remains the necessity of specific implementation for each concrete NoSQL language, as well as the lack of an example of complete integration between concrete SQL and concrete NoSQL database. This was examined in the following papers.

One of the first solutions to the problem of integrating SQL databases with other data sources was suggested by the authors (Melton et al., 2002). Their SQL/MED framework offered an extension of SQL and enabled integration with external data sources (called "non-SQL data sources" by the authors), which include external data files, XML documents, and more. Authors (Roijackers and Fletcher, 2013) also examined the problem of integration of different type databases, mostly from the aspect of parallel readings of data from SQL and NoSQL databases. They developed a prototype frame and mapper for transforming NoSQL queries into SQL queries and ensured the display of the data obtained from the NoSQL database in the form of a relation. The mapping between NoSQL and SQL, as well as between SQL and XDM (XML Documentation Mappings) concepts was the subject of work (Valer et al., 2013), all of it with the aim to create a basis for XQuery language application (i.e. XML query language) in NoSQL databases. In the paper (Curé et al., 2011) the difference between imperative languages of most NoSQL databases and declarative SQL language was solved by the introduction of mapping language by which the concepts of the source database (regardless whether it is SQL or NoSQL database type) are mapped into a global schema. The obligation that even SQL queries have to be translated, as well as the absence of support for more complex queries with JOIN clause, represent the biggest weakness of this approach. Some authors (Sellami et al., 2014) advocate the introduction of a dedicated integration level. They suggest REST user interface called ODB API, which enables an equal execution of CRUD operations of SQL query language over SQL and NoSQL databases, which was demonstrated on the example of Riak and CouchDB. ODB API does not support complex queries or the use of a JOIN clause, which in turn is its biggest flaw. Another suggestion of an integration layer, called Sinew, is presented in a paper (Tahara et al., 2014). Sinew represents a layer which also enables carrying out of SQL statements over SQL and NoSQL databases. This layer enables parallel utilization of the data obtained from SQL and JSON. Authors (Alomari et al., 2015) suggest a joint data model and universal user interface for SQL and NoSQL databases. The usability of the approach is demonstrated on the integration of document-based databases (MongoDB and Google DataStore) and column family-based database (SimpleDB). Lawrence in his work (Lawrence, 2014) examined the problem of integration in general, but also from an aspect of the integration of different types of NoSQL databases, which instead of SQL use their own languages. This author suggested the introduction of an additional level for uniform use, called Unity. With Unity, he expanded the three-tier architecture with a new layer, which he treated as a virtualization layer. The main characteristic of this solution is that databases of different types are used for storage of data of different structuredness level. That is a desirable characteristic, as it allows a database of a suitable type to be used for an appropriate purpose while allowing the user to use uniformed SQL language to work with both types of databases.

Table 1
Statement mapping rules between representative languages of different types of databases.

Language	*SQL*	*SPARQL*	*Cypher*	*CQL*	*UnQL*	*Mongo QL*	*REDIS QL*
Model	Relational	Graph	Graph	Column family	Document	Document	Key–value
Insert data	*INSERT*	*INSERT*	*SET*	*INSERT*	*INSERT*	*insertOne()* *insertMany()*	*SET*
Update data	*UPDATE*	*MODIFY*	*SET* and *FOREACH*	*UPDATE*	*UPDATE*	*updateOne()* *updateMany()*	*SET*
Delete data	*DELETE*	*DELETE*	*REMOVE*	*DELETE*	*DELETE*	*deleteOne()* *deleteMany()*	*DEL*
Select data	*SELECT*	*SELECT*	*RETURN*	*SELECT*	*SELECT*	*find()*	*GET*
Source of data	*FROM*	*(FROM)* *URI*	*MATCH*	*FROM*	*FROM*	.collection	/
Filtering	*WHERE*	*WHERE* */ FILTER*	*WHERE*	*WHERE* (index column)	*WHERE*	{*field1*:*value1*}	/
Join operation	*JOIN*	Using pattern	*MATCH* (approximately)	/	/	*$lookup and "."* *(for nested)*	/

Contrary to creating and using a joint model, which was characteristic of previous approaches, the polystores approach is based on the observation that "one size does not fit all". The paper (Duggan et al., 2015) defines the basic postulates of this approach that other authors have taken as their starting point and expanded in their works (Kharlamov et al., 2016; Dasgupta et al., 2016; Maccioni et al., 2016; McHugh et al., 2017). The authors (Duggan et al., 2015) highlight support for unifying querying over multiple data models as an important feature. This paper describes a BigDAWG prototype made up of SciDB, Accumulo, Postgres and S-Store database management systems. BigDAWG was created in order to achieve three goals: location transparency, semantic completeness and to provide users with the ability to query the same data using different languages. In the proposed architecture, these authors use the island of information, which represent "a group of storage engines addressed with a query language" (Duggan et al., 2015) and shims, which are responsible for translating user-entered commands into the language of the destination database. According to the authors, one island can have multiple shims to different databases in order to maximize load balancing and optimize execution. In order to achieve semantic completeness, the authors introduced degenerate islands in charge of a particular type of storage system (relational, array, etc.), which made the proposed architecture more complex. When adding a new storage system, it is necessary to create a new shim for each island that accesses that storage. If a new type of database is introduced, it is also necessary to create a new degenerate island.

The polystores are based on multiple data models that cannot be uniquely administered. This particular characteristic affects the inability to control and reduce the redundancy, which the authors (Duggan et al., 2015) themselves noted and listed as a direction for further research. In addition, the multiplicity of models also influences difficult coordination and synchronization (among storage systems of different types), which are important indicators of integration (Niu et al., 2013). Also, the complexity of the architecture by adding new shims (degenerate islands if necessary) for each type of storage system and increasing the number of "communication channels" between the architecture components negatively affect the simplicity of the solution, as one of the indicators of usability (Niu et al., 2013). The polystores approach offered an innovative solution for connecting different storage systems. However, given that integration and usability were taken as key NFRs in this paper, the polystores and their architecture cannot be applied as an adequate solution for the scope of this paper. Similar to the polystores, the other analysed approaches have also brought innovations in the form of modification or extension of the three-tier architecture and opened the space for further research and improvements.

A shared feature of works analysed in this Section is that their authors, no matter if they explicitly use the term hybrid databases, investigate the challenge of integration and uniform use of different type databases, which represent the essence of the hybrid database approach. Hybrid SQL/NoSQL database consists of a SQL database and one or more NoSQL databases of a different type which are treated as components of a single logical database. Also, because of a single data model, a hybrid database provides a user with a sense of simplicity of working with a single database, instead of working with multiple individual databases. At the same time, it enables him to use all the specific storage structures that the hybrid components possess. This makes the end-user relieved of the need to take account on physical locations where the data is and what component of hybrid and what specific storage structures he uses, while allows him to see and utilize the entire hybrid database as a unique logical database. Due to all of this, the hybrid databases were chosen as a starting point while designing a new architecture for integration and uniform use of different type databases.

Improvements, in which new architecture should extend the functionality of existing solutions, may be formulated in the following fashion:

- Complete integration (by centralizing the process of administration of all databases with controlling of storage structures (types of objects), constraints, integrity rules and redundancy for the entire logical database)
- Enhancement of usability (by enabling the mapping of the chosen universal language (SQL) into all specific languages of the destination databases while supporting data joining from all components)

In most of the analysed works, uniform use of different types databases was achieved by using SQL language (as a standardized and widespread language for working with SQL databases Winand, 2017), with the definition of mappers for translation of SQL queries into concrete languages. Those languages are used by specific databases, which do not support the SQL standard. The absence of standardization of languages for work with NoSQL databases in general (as well for NoSQL sub-types) suggests that SQL language is also applicable as the integration language between SQL and NoSQL databases. Because of this, SQL was chosen as the uniform language in the newly developed architecture as well.

The process of integrated administration of components of a hybrid database can be realized with joint administration of storage structures, constraints, and integrity rules, with a tendency to reduce redundancy on the level of the hybrid database in general as well as on the level of single components. This makes data from the NoSQL database susceptible to some safety mechanisms of SQL components, which contributes to further integration of hybrid SQL/NoSQL database components. The spotted possibility for further integration of SQL and NoSQL databases and their uniform usage as components of a unique logical database have been used as guidelines in the development of the new architecture for hybrid SQL/NoSQL database, described in the next part of this paper.

3. The proposed architecture for SQL/NoSQL integration and uniform use

In order to achieve a uniform use, the user needs to be provided with a single language for all the statements executions over the hybrid, regardless of which component is used. The integration should relieve the user from the need to take account of the database type where the data is stored, which structure is used for storing and in what way the data from different components are joined and integrated into the result. From the aspect of the user, usage of a hybrid database should provide simplicity in working in a way that the user executes statements over a single DBMS. The basic way of executing statements over any DBMS may be described by the following use case scenario:
Utilizer: System user
Use case name: Statement execution
Precondition: The user has started the form/interface to enter the statement
Basic scenario:

(1) The user enters the statement.
(2) The user initiates the statement execution.
(3) The system executes the statement and displays the result of the statement execution.

The first two events represent the user's actions, whereas the third is the system's response. The term "system" generally implies an information system, which contains the database of the appropriate type (SQL or NoSQL). With the aim of retaining the simplicity of work, the actions of the user, stated in the aforementioned use case, are kept in the description of the use case of hybrid database use as well. Thus, when using a hybrid database from the aspect of the user, the same degree of simplicity is retained as if the SQL database is used. By including specific guidelines for hybrid database into the first use case, a new use case of statement execution in the hybrid database is formed. The description of the basic use case scenario for working with hybrid database follows:
Utilizer: System user
Use case name: Statement execution
Precondition: The user has started the form/interface to enter the statement
Basic scenario:

(1) The user enters the statement.
(2) The user initiates the statement execution.
(3) The system decomposes the statement into sub-statements, determines in which components the suitable sub-statement are to be executed using which data management language.
(4) The system adjusts the entered statement and executes it on a specific language of the database of a specific type.
(5) The system accepts the results of the execution of each component, unifies them, translates them into the user's language and displays an integrated result.

Examples of use cases that demonstrate integration and uniform use, which are achieved through the use of newly developed architecture, are given in the next section. The basis of the presented architecture is made up of widely spread three-tier architecture, which is extended by specific components for enhancement of integration and usability. The display of the architecture, which supports integration and uniform use of components of a hybrid SQL/NoSQL database, is shown in Fig. 1.

The first level, just like in the traditional three-tier architecture, contains the user interface, which serves as the accessing point of the system. The interaction between the user and the system, based on the new architecture and using a hybrid database, is to be realized by SQL query language. Components are added to the new architecture, which analyses and adjusts the SQL statements execution over all database types, which are contained in the hybrid database. Even though there are different types of user interfaces in the aim to show the use of the new architecture, their implementation specificities are not of interest, as they do not influence the execution logic. Thus, we introduce the simplification of this layer, as the plain interface where SQL statements are entered, and which display the result of their execution, i.e. we treat it as SQL console.

SQL API (Application Programming Interface) is in charge of communication with the presentational layer, and it represents a part of the newly developed wrapper layer. This API supports the SQL standard and accepts the entry of statements created in the mentioned language. Besides this, the wrapper is in charge of managing statement execution process, from the moment when it accepts the statement entered by the user, to the moment when the user sees the results of the statement execution, in other words, it has a role of an execution controller.

After accepting the SQL statement, the wrapper is in charge of its analysis, processing and directing towards the destination database for execution. Even though some developers of NoSQL database management systems strive to develop SQL query language support (Grolinger et al., 2013), SQL language is not directly applicable to all NoSQL database types, nor to all NoSQL database management systems. This poses a need to translate the entered SQL statement into a specific query language of a concrete DBMS (if destination DBMS does not support SQL), which cannot be conducted without previously defined translation rules. In order for this to be realized, it is necessary to analyse and decompose the entered statement, and determine which database type, and with it which DBMS, poses the source of the desired data reading, or the destination of data inserting, updating or deleting. For the purpose of detecting the type of destination database which owns the objects of the entered statement and with the purpose of adequate management of further query execution (decomposition of the same and optional translation into specific languages) the wrapper layer includes specific components, controls the communication between them and integrates their work. Those components are:

- Entered Statement Processing Component (ESP),
- Key Words Search Component (KWS),
- Constraint Controller (CC),
- Statement Mapper (SM) and
- Integration Controller (IC).

3.1. Entered Statement Processing Component (ESP)

In order to determine the type of destination databases, the entered query is firstly analysed in the wrapper layer. This functionality of the wrapper is realized in the Entered Statement Processing Component (ESP). By dividing the statement into clauses,

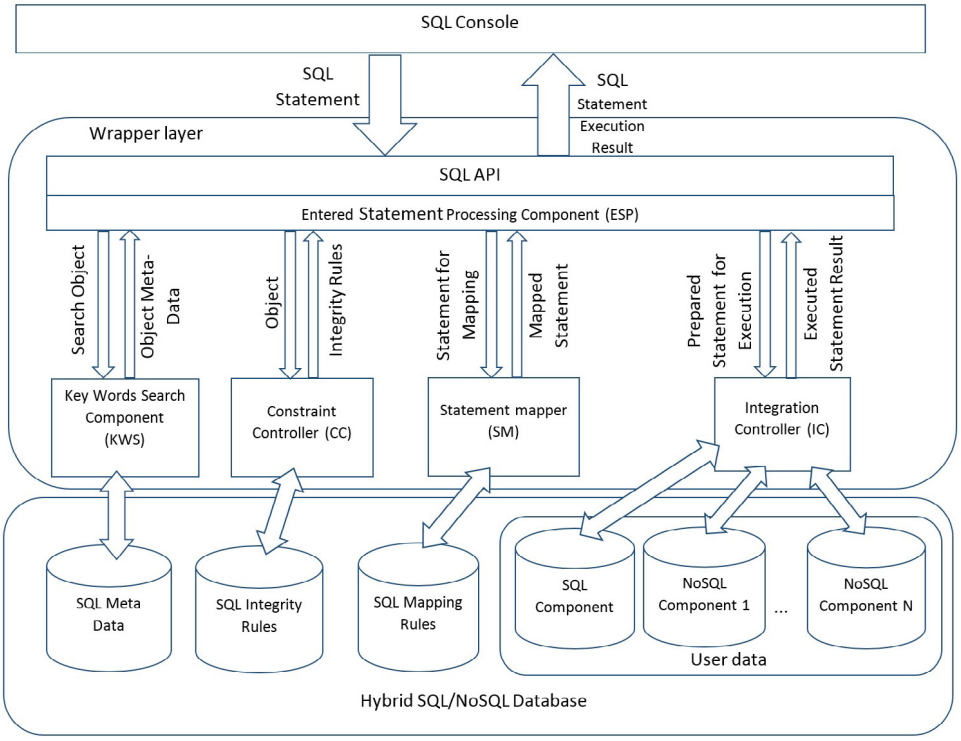

Fig. 1. The graphic display of the architecture for integration and uniform use of components of hybrid SQL/NoSQL database.

ESP commences the "processing" of the entered SQL query. In the initial phase of the query "processing", FROM clause is of the special significance. It contains the names of the tables, views or functions, which are significant for the execution of the query. Since after the word FROM we most commonly see the word "table", representing the so-called "TableExpression" (Oracle, 2019b), in the following text, a table will be implied as to the common source of data. The idea is that ESP supports the syntax of table joining in the traditional way, which divides the column by the "comma" symbol (,) as well as the syntax according to American National Standards Institute recommendation, according to which, keyword JOIN is used for connecting tables within the FROM clause. In both cases, ESP is able to recognize and pull out the table names from the entered statement. The user is provided with the possibility to write SQL queries without minding logical or neither physical data structure nor their organization in the hybrid database. By using SQL syntax, the user accesses all the resources of all databases in a uniform way as parts of a unique hybrid database. ESP needs to adjust the entered statement for execution, which depends on destination databases' types where the statement and its segments will be executed. Besides this, the specific challenge the ESP component encounters in the course of statement processing is the possibility that one query accesses the data from different objects, which belong to different database management systems. In order to overcome the challenge, it is necessary to search for answers to the following questions:

- How to organize and manage data on object types, on database types they belong to and on specific languages they use?
- To what extent is it possible to manage the constraints in the hybrid SQL/NoSQL database made up of heterogeneous types of databases?
- In what way to conduct the translation of the entered statement into the suitable statement of the destination database's specific language?
- In what way to join the data from the databases of different types within the hybrid SQL/NoSQL database and how to integrate the obtained results of the statement execution over different components of the hybrid into an integral result which is to be displayed for the user?

In order to answer these questions and overcome the challenges the questions have arisen from, the ESP component communicates with other dedicated components, forwards statements to them, processes the returned results and integrates the execution of those components functionalities through managing the entire process of realization of the entered statement. Those newly developed components the ESP communicates with are:

- Key Words Search Component (KWS) – to obtain meta-data on the statement's objects (object type, database type where it is stored and similar),
- Constraint Controller (CC) – to obtain the data on integrity rules defined for the statement's objects,
- Statement Mapper (SM) – to obtain the translated (mapped) statement of the specific language of the destination database (if the destination database is not an SQL database or a database which supports SQL language) and
- Integration Controller (IC) – which is used for forwarding the statements, which are to be executed, and which will obtain the executed statement's results.

3.2. Key Word Search Component (KWS)

In order to adjust the statement to the destination database, the ESP component communicates with a component called Key Words Search Component (KWS). This component uses the data from the meta-data dictionary, which has been extended to meet the needs of the new architecture. Besides the usual object data, the meta-data dictionary in the new architecture also contains specific data on hybrid database objects:

- Object type (table, view, document, key–value pair, column and so on),
- Database type it belongs to (SQL or a concrete NoSQL database sub-type: database based on key–value pair, on a document, on a column family or on a graph),
- Concrete DBMS it belongs to (Oracle, PostgreSQL, MongoDB and similar).

This dictionary eliminates the need for the user of the hybrid database to know or take account on how the data is stored, what structures are used for its storage, which database type contains the data, in what concrete DBMS it is and what language the DBMS uses. Meta-data dictionary contains the necessary information on the data's organization, which enables the user, by only typing in SQL statement, to access all the hybrid database data in a uniform way, regardless of the data are stored in one or more components of the hybrid. When the time comes for the entered statement transformation or a transformation of its parts into a specific DBMS language, the KWS component will read data from meta-data dictionary, filtered by the table names from the initial SQL statement. That way the KWS component represents the link between the ESP component, which sends to the KWS names of the objects from the statement, and the meta-data dictionary. From the meta-data dictionary, the KWS retrieves the object type, the database type and the concrete DBMS of objects from the entered statement. KWS returns the loaded meta-data to the ESP component, which analyses them and makes the decision on whether it is necessary to transform the SQL statement into the destination database language. Bearing in mind the two potential outcomes of the execution of the entered statement, the ESP uses the following guidelines in any query realization:

(1) If the query is executed over the destination database, which supports SQL syntax, translation does not happen. Before the query is executed, the integration rules are checked. If necessary, the query shall be supplemented by suitable filter criteria or a new sub-query is to be generated,
(2) If the query is executed over the database, which does not support SQL, the entered query shall be translated into the destination database language, but, before the mapping and the execution of the statement, the integration rules will be checked. Should there be the need, the query might be supplemented by suitable filter criteria or an additional query will be generated.

3.3. Constraint Controller (CC)

Constraint Controller is in charge to check constraints, as well as the conditions for joining objects from databases of different types. The base for object join, regardless of which hybrid's component they belong to, is made up of entity integrity rule (stating that no primary key attribute can have null value) and the referential integrity rule (defining the way primary and foreign keys are connected). These rules, specific to SQL database, have been chosen as the basis for hybrid SQL/NoSQL database constraints as SQL database provides a higher level of constraint control than NoSQL database does (Nance et al., 2013).

By adjusting the two integrity rules to the needs of the hybrid database, two ways of their realization have been defined. Each hybrid's object, no matter the database type it is in, complies with at least one of these two conditions:

- The object contains a reference to one or more other objects over reference attributes (foreign keys or attributes which play that role) no matter in which component of hybrid database those objects are and
- The object identifier (primary key of a table or an attribute of another object type, which plays that role) is referenced by at least one object from the same or some other database, which represents a component of the hybrid database.

Joining objects within the SQL database does not pose a problem due to their support of integrity rules in SQL databases. However, when joining objects that belong to databases of some NoSQL type, a new challenge appears. That is because NoSQL databases do not support direct use of integrity rules of SQL databases. The role of overcoming the challenge has been delegated to the ESP component. The ESP component demands from the component named Constraint Controller (CC) meta-data for desired objects in order to check the integrity rules. The CC component reads data from the dedicated table (*INTEGRITY_CONSTRAINTS*) and sends them to the ESP component. For each object and for each relationship the object builds with another object in the model a single entry in the *INTEGRITY_CONSTRAINTS* table is defined, containing the following data:

(1) Object name (unique on the level of the hybrid database),
(2) Integrity rule type: entity integrity rule ("*ENT_INTEGRITY*") or referential integrity rule ("*REF_INTEGRITY*"),
(3) Object identifier: column (or columns) of a primary key for SQL databases or the appropriate field (or fields) which is to take that role with NoSQL database object, and which must not have a null value,
(4) Referencing property: a foreign key column for SQL databases or a responding field which is to take that role with NoSQL database object,
(5) Name of the objects which is being referenced by the referential integrity rule,
(6) Name of the column or the field of the referenced object.

The entry number (3) of the described structure satisfies the entity integrity rule (as the primary key cannot have null value) while entries from (4) to (6) enable the use of the referential integrity rule. In the case of entity integrity rule register (Table 2), column (2) has the value "*ENT_INTEGRITY*", column (3) is filled in with appropriate identifier name whereas the columns from (4) to (6) remain blank. The entry of this structure is entered into the *INTEGRITY_CONSTRAINTS* table, no matter whether it is a SQL database object (table) or an object of some NoSQL database subtype.

In the *INTEGRITY_CONSTRAINTS* table, the columns' header contains a general term "*OBJECT_NAME*" as this table contains integrity rules for both SQL components objects (tables) and NoSQL components objects (documents, key–value pairs, column families, graph nodes). For the same reason, the term "property" is used (including columns, fields, etc.).

In the second case (Table 3), when within the *INTEGRITY_CONSTRAINTS* table, referential integrity rule is registered, the table entry contains values for the columns (4) to (6) as well. The repeated value of object name and its identifier name in multiple syllables for an object, which contains more integrity rules, represents a controlled redundancy and enables a single database table (*INTEGRITY_CONSTRAINTS*) to contain entries of both rule types.

Since the mentioned meta-data are clearly structured and that it is necessary to ensure their consistency and durability, the table *INTEGRITY_CONSTRAINTS* is placed in the SQL type of database. For storing the meta-data on integrity rules, the new architecture has enabled the design of a dedicated SQL database instance. This separated the user data storage and integrity rules storage (containing meta-data), which ensures their independent administration, more flexible management of privileges and

Table 2
The look of *INTEGRITY_CONSTRAINTS* table and the suitable entry for *PRODUCT_DECLARATION* object (entity integrity rule).

OBJECT_NAME	RULE_TYPE	OBJECT_IDENTIFIER	REFERENCING_PROPERTY	REFERENCED_OBJECT	REFERENCED_PROPERTY
PRODUCT_DECLARATION	ENT_INTEGRITY	DECLARATION_ID	(null)	(null)	(null)

Table 3
The look of *INTEGRITY_CONSTRAINTS* table and the suitable entry for *PRODUCT_DECLARATION* object (referential integrity rule).

OBJECT_NAME	RULE_TYPE	OBJECT_IDENTIFIER	REFERENCING_PROPERTY	REFERENCED_OBJECT	REFERENCED_PROPERTY
PRODUCT_DECLARATION	REF_INTEGRITY	DECLARATION_ID	PRODUCT_ID	PRODUCT	PRODUCT_ID

better recovery. The same principle is applied to the matter of mapping rules and meta-data dictionary. Taking all this into account, in Fig. 1, the three dedicated repositories (mapping rules, meta-data dictionary and integrity rules) which are of SQL database type are shown as a part of data layer, but, at the same time, also as separate SQL database instances whose data are separated from the user data. On the graphic depiction, it is clearly marked which component communicates with which repository (KWS with the meta-data dictionary, SM with mapping rules, CC with integrity rules).

In the course of statement execution, and in order to satisfy the integrity rules, it is necessary to compare whether the entered values of foreign keys of the referencing object are in compliance with values of primary keys of referenced objects. Since the data can be stored in a database of different types, the control is completed in the ESP component, which uses the data from the *INTEGRITY_CONSTRAINTS*, sent by the CC component. On the basis of the received data, it is confirmed whether the entered values are in compliance with existing values of the referenced object and whether a value is entered for an object identifier. The ESP's functions, which enforce the integrity rules, are:

- Additional query generation and
- Existing query modification.

Additional queries shall be generated for checking the concrete values, which must be in accordance to reference integrity rule. For example, when the data is entered into an object, which references another object, the ESP component will generate a query, which will check if the entered value of the property of the foreign key suits some of the values of the properties, which serves as the primary key in the referenced object. The modification of the query is executed when it is necessary to join data from different objects, which belong to databases of different types. The greatest challenge, in this case, is the JOIN clause. As it cannot be directly applied to the objects from databases of different types, the ESP component divides the entered query into smaller queries so that each of them has one destination database. The ESP component receives the data on the integrity rules of each of the objects from the CC. Based on the received data, it generates new clauses of the newly-obtained queries. Thus, for example, if it is necessary to display data from the SQL table, which references a document of NoSQL database, the ESP component shall modify a part of the SQL query, which is executed over the NoSQL database. That part of a query is modified in a way that inside it those entries from SQL database, which have suitable pairs from NoSQL query, are filtered. This way, the ESP component satisfies the integrity rules received from the CC component. Practical examples of the usage of the new approach and its components on examples from real life are described in the next section.

3.4. Statement Mapper (SM)

For adjustments and translation of the user SQL query into the specific language of a concrete DBMS, the ESP communicates with the Statement Mapper (SM). This communication is preceded by communication between ESP and the CC component. After the analysed integration rules which the ESP component has obtained from the CC, the ESP "turns to" the SM component. The SM component, as its name suggests, uses mapping rules. The mapping rules determine the way SQL statements (SELECT, INSERT, UPDATE and DELETE) are transformed into a syntax of the destination database language. The mapping rules are defined depending on the statement type, database type and the language used by the concrete DBMS. The mapping rules between SQL and languages of popular database management systems, which do not support SQL, are shown in Table 1.

Statement mapper function may be described in the following manner: the ESP forwards the query, object name, object type, database type and concrete DBMS to the SM component. Based on the entry, the SM component translates the entered query by loading the transformation rule of each clause from predefined rules and adjusts specificities (operator and alike) to the destination database. Having in mind that by using the SM component new query syntax is defined, but that the queries themselves are not executed in it, this component shall return the "translated" query to the ESP component after the transformation. During the execution of queries, whose destination databases are SQL and NoSQL components of hybrid, the resulting data, if there are no additional conditions, shall first be selected from the NoSQL component (by the execution of query part which refers to this component, created by ESP). Then, the ESP shall replace the resulting NoSQL data in the WHERE clause of the query executed over SQL component by obeying the integrity rules. This reduces the number of statements necessary for mapping. The described communication by which the SM component returns the modified query to the ESP component additionally emphasizes the role of the controller, which the ESP component bears during the entire process of working with the entered statement, from the moment of its entry to the moment when the result is displayed to the user.

3.5. Integration Controller (IC)

Integration Controller (IC) is a component in charge of statement execution over the hybrid database. The statements which are to be executed over hybrid database components are given to the IC by the ESP. The IC component establishes a connection over the destination databases (if it is not open) and executes suitable statements. Which statements shall be executed over which component of the hybrid database is decided by the ESP. Furthermore, after the statements are executed over a database, the results returned from the IC component are processed by the ESP component. The ESP analyses the results, and, if necessary, formats and connects the statement (typically queries) execution results over different components of the hybrid database into an integral result, which shall be displayed to the user via the SQL console, in the form of results of the executed SQL statement. This rounds up the entire process of statement execution over a hybrid database. Introduced components (KWS, CC, and

SM) are also good candidates for other integration problems. For example, SM component can be used for mapping statement not only between hybrid components languages, but also when integration and unification of different NoSQL databases are set as requirements, either inside certain sub-type (documents, graphs and others) or among them. Also, SM component with certain modification can be used for mappings even between SQL procedural extensions (e.g. PL/SQL and T-SQL). Afterwards, if users have needs for strengthening up constraints for any NoSQL database, CC component could be an interesting option. KWS component expands regular meta-data with some additional information and with certain adjustments is applicable for advanced searches of database dictionary.

4. Business cases and examples of new architecture usage

The new architecture usage, as well as the effects of integration and uniform use of components of a hybrid database, which are enabled by this architecture, will be presented on a chosen business example. The selected example represents a part of Product Life-Cycle Management (PLM) function, which is a common function in production organizations, and, among others, contains records of products and their declarations. The selected example is displayed as a conceptual UML class diagram in Fig. 2. In order to eliminate classes and attributes, which are irrelevant for this example, the technique of abstraction is used. The product's entity holds more or less unchangeable property structure, as the company keeps a record of basic attributes of all its products (such as the product ID, full product name and its shorter name). The entity representing product declaration may show some deviations of property structure, even though all manufacturer declarations have a well-known purpose. For the observed system a declaration is an external document whose exact structure is defined outside of the system (e.g., some declarations may, but do not have to, display attributes such as description, material type, contents, dimensions, bearing capacity, energy source and similar, which is determined by product's type). The named characteristics of products and declarations were guidelines in the course of decision making on the distribution of the listed entities to the exact hybrid database component. Thus, the product is allotted to the SQL component, whereas declaration is placed in the NoSQL component (which is document-based, because of the declaration's natural feature to present a document). Oracle DBMS is used as a SQL database representative, while MongoDB DBMS (MongoDB, 2019) is chosen to be a document-based NoSQL database representative. At the time of the selection, the two database management systems were the best-ranking representatives of SQL and NoSQL database, respectively (SolidIT, 2019).

In order to show new architecture usage, four use cases were selected, covering all four CRUD operations. Use cases whose operations have an SQL component as a destination database are not shown in this work because of their triviality (since they do not require the mapping of the initial SQL command). In contrary to them, the selected use cases are more complex because each of them requires statement mapping and the usage of newly developed architecture components:

(1) Data deletion (destination database is NoSQL component of hybrid SQL/NoSQL database)
(2) Data updating (destination database is NoSQL component of hybrid SQL/NoSQL database)
(3) Data insertion (destination database is NoSQL component of hybrid SQL/NoSQL database)
(4) Display of integrated data from SQL and NoSQL components of a hybrid SQL/NoSQL database

Although SQL syntax for simultaneous updating of various tables can be found, some of the largest DBMSs vendors (like Oracle) do not directly support this syntax (and instead provide this functionality either through multiple operations of a single transaction or procedural extensions). Therefore, a simultaneous update of several tables is set for the limitation of this paper as well as for the future direction of research. A more advanced use case was chosen to display the data when the data is selected from different hybrid components. In addition to the mapping in this example, the logic of merging data from different hybrid components will be explained as well.

4.1. Deleting data by using new architecture

The first use case deletes data from NoSQL component of hybrid. The example of entered statement for deleting *PRODUCT_DECLARATION*, with *DECLARATION_ID = 100* is:

```
DELETE FROM PRODUCT_DECLARATION WHERE
DECLARATION_ID = 100;
```

After accepting the statement, the ESP component determines the object name from which the data will be deleted. With joint work with the KWS component, they reach meta-data for *PRODUCT_DECLARATION* object. In the given example, as it was already mentioned in the conceptual model description, *PRODUCT_DECLARATION* object is of the document type, belonging to the NoSQL database type, which is document-based and stored in MongoDB DBMS. The next thing to be inspected is whether the *PRODUCT_DECLARATION* object has been referenced by some other objects. Since the *PRODUCT_DECLARATION* does not have child records (which is checked by CC component), inserted statement proceeds to statement mapping. Using SM component ESP gets translated statement:

```
DB.PRODUCT_DECLARATION.DELETEONE
({DECLARATION_ID:100});
```

ESP component forwards translated statement to the IC component. IC component executes the statement, after which it sends results to the ESP components, which is responsible for displaying data to the user.

4.2. Updating data by using new architecture

Similar logic is applied for update statement on a concrete object. For example, if a user wants to update the *DESCRIPTION* property of PRODUCT_DECLARATION with *DECLARATION_ID = 200*, he will enter the statement:

```
UPDATE PRODUCT_DECLARATION SET DESCRIPTION =
'Broad use product' where DECLARATION_ID = 200;
```

After the KWS component detect that the destination object is a document of document-based NoSQL database, precisely of MongoDB, the EPS components send a request to the CC component for constraint check. In this example, there is no update of any foreign key, which implies there is no restriction in update execution. The following example (use case for insert) contains constraint of integrity rule and shows how it must be satisfied when executing a statement. After mapping statement, the MS component returns to the ESP following statement:

```
DB.PRODUCT_DECLARATION.UPDATEONE
({DECLARATION_ID:200}

{$set:{ DESCRIPTION: "Broad use product"}});
```

The mapped statement is returned to the EPS, after which is sent to the IC for execution.

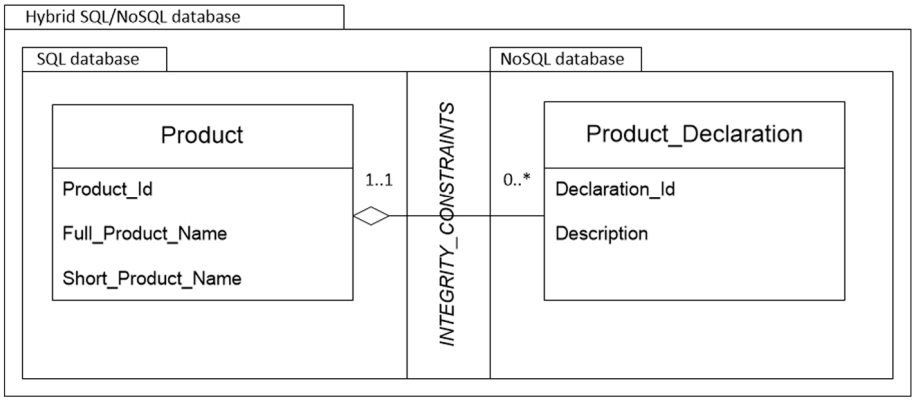

Fig. 2. A part of conceptual UML class diagram for Product Life-Cycle Management (PLM).

4.3. Inserting data by using new architecture

For the presentation of inserting operation, the use case on *PRODUCT_DECLARATION* was selected. The statement for inserting *PRODUCT_DECLARATION*, which has the value of 123 for *DECLARATION_ID*, the value of 111 for *PRODUCT_ID*, and description (*'Broad use product'*) would state:

```
INSERT INTO PRODUCT_DECLARATION (DECLARATION_ID, DESCRIPTION, PRODUCT_ID) VALUES (123, 'Broad use product', 111);
```

Similarly to the previous two examples, ESP and KWS components retrieve meta-data about document named *PRODUCT_DECLARATION*. This step is essential because based on the meta-data ESP decides whether the data will be entered into the hybrid or not. The rule that the object over which data is entered must exist is defined to prevent data entry into a non-existing object. With SQL commands this is not a problem, but NoSQL databases (like MongoDB) automatically create a collection when inserting data into a so far non-existing object. The problem is that mistype object name may cause the creation of a new collection, which is certainly not desirable. One of the directions of development is the introduction of support for DDL commands, so the use of these commands will make it possible to create the objects in a controllable manner.

The given model (Fig. 2) indicates that for the insert of every *PRODUCT_DECLARATION*, it is necessary to insert the value for the attribute *PRODUCT_ID*, which represents a foreign key to *PRODUCT*. This is defined by the integrity rules, more precise by the referential integrity rule, as already shown in Table 3. The referential integrity constraint for this example is significant as it prevents a *PRODUCT_DECLARATION* from being inserted for a *PRODUCT* which does not exist or which has not yet been inserted into the *PRODUCT* table. In order to check integrity constraints fulfilment, the ESP component will create an additional SQL statement, which is to inspect the value of *PRODUCT_ID*, entered by the user. In the described use case, the reason for generating the SQL statement is that the *PRODUCT* data are placed in the SQL hybrid database component. The generated statement for referential integrity inspection would be:

```
SELECT 1 FROM PRODUCT WHERE PRODUCT_ID = 111;
```

If the entered value for *PRODUCT _ID* (in this case 111) has a suitable pair in the *PRODUCT* table, the result of the query execution will return the control value (in this example, it is literal "1") and data insert in the *PRODUCT_DECLARATION* document may be executed. For the operation to be conducted, it is necessary to map the SQL insert statement into a suitable insert statement of MongoDB language. The transformed statement shall have the following syntax:

```
DB.PRODUCT_DECLARATION.INSERT({DECLARATION_ID: 123, DESCRIPTION: "Broad purpose product", PRODUCT_ID:111});
```

After the statement mapping, it is executed over the NoSQL component of the hybrid SQL/NoSQL database, or, more precisely, over the MongoDB component of hybrid. This reflects the uniformity of the new approach in working with all components of the hybrid database, as the user will complete the data insert into the NoSQL component in the same way as he would have done into the SQL component.

4.4. Display of data with the new approach architecture

The next example represents a use case where the resulting data are simultaneously retrieved from different objects and from databases of different types (more precisely SQL and document NoSQL), which represent components of the hybrid SQL/NoSQL database.

The selected use case needs to display all the data on products and their declarations. The suitable SQL statement would read:

```
SELECT * FROM PRODUCT JOIN PRODUCT_DECLARATION USING (PRODUCT_ID);
```

In the concrete example, the *PRODUCT* object presents a SQL database table (Oracle DBMS), whereas the *PRODUCT_DECLARATION* presents a document within the NoSQL database, which is document-based (MongoDB). This implies that a part of the query for the *PRODUCT* table will not need mapping into the destination database language. That part of the initial statement is:

```
SELECT * FROM PRODUCT;
```

For the *PRODUCT_DECLARATION* document, it will be necessary to conduct mapping into the destination database-specific language. However, before the transformation of the query, the CC shall access the integrity rules according to the ESP component demand. The entry from Table 3 also suits the referential integrity for this example. In order to join adequate entries from the *PRODUCT* table with the data from the *PRODUCT_DECLARATION* document, it is necessary to obey the mentioned integrity rule. This is achieved by executing one part of the query and replacing its results into the second query part. If the WHERE clause concerning the NoSQL object is not present in the entered statement, as it is the case with the currently analysed query, then the part of the entered query aiming NoSQL component is executed first. This rule of the new approach is defined in order to reduce the number

of clauses, which will be mapped into the specific language of destination NoSQL database. This directly saves time necessary to map the query part into the specific query of the destination database. The adequate statement, which is to be executed in MongoDB DBMS is:

```
DB.PRODUCT_DECLARATION.FIND({});
```

The results of the named query being executed are returned to the IC component, which forwards them to the ESP component. The ESP extracts values for *PRODUCT_ID* from received results and translates them into SQL table format which is dynamically added to WHERE clause of the SQL query, which needs to be executed. After the ESP modifies the initial SQL query, the resulting query which is to be executed over the SQL component, written in pseudo-code, shall read:

```
SELECT * FROM PRODUCT WHERE PRODUCT_ID IN
LIST_OF_VALUES_FROM_NOSQL_QUERY;
```

After this query execution and the selection of only those products, which have at least one declaration, the IC conducts join operation of the resulting SQL query table with the results of NoSQL query execution. The results of NoSQL query execution were translated in the previous step into the SQL table form, adding now, besides the *PRODUCT_ID*, the remaining fields retrieved from the NoSQL query to the resulting SQL table. This enables the join of the data from the two components and their integration into a unique result in accordance with the defined referential integrity rule. These examples demonstrate the new approaches uniformity and integrating functionality when working with different statements over different components of the hybrid database.

5. Experimental results

The use cases described in the previous section were executed over the architecture presented in this paper. It is important to emphasize that this is still a prototype. Not all the functionalities have been fully implemented yet. Nevertheless, the tests were conducted in order to show the feasibility of executing use cases containing CRUD operations and to display a potential impact on performance, i.e. statement execution time. Data display tests have demonstrated the joining of data from the SQL and NoSQL hybrid component, which represents one of the important functionalities of the new architecture and one of the important contributions of this approach. The use cases were tested on a system that has an Intel i7 2.9 GHz processor with 16GB of RAM and an SSD. Entering and executing commands, as well as measuring elapsed time, were implemented in the NetBeans IDE. Oracle 12c DBMS was selected for the SQL component of the hybrid database, while MongoDB 3.6 was used for the NoSQL component. These DBMSs were selected because they represent the most popular SQL or NoSQL databases nowadays (SolidIT, 2019). Along with executing statements on a hybrid SQL/NoSQL database, they were executed individually on Oracle and on MongoDB, in order to compare their results with those achieved by hybrid. Tests for CREATE, UPDATE, and DELETE were realized over 10, 100, 1000, and 5000 records. Due to the natural property of a READ operation, which typically accesses a larger number of records during one run, the number of records was expanded by a set of 50,000 and 100,000 in addition to the above.

All operations execution were performed with 10 repetitions, after which the average values of execution time shown in Fig. 3 were calculated. It took average 11, 68, 1053, and 3813 ms for SQL database (Oracle) to execute use case for inserting data with 10, 100, 1000, and 5000 records respectively. Meanwhile, average execution times for NoSQL database (MongoDB) were 23, 70, 679, and 1756, while for hybrid SQL/NoSQL database the average results were 67, 105, 794, and 2792 ms respectively. Based on this data, a graph was generated (Fig. 3a). Fixed costs of establishing a connection and loading of required resources gave the SQL database an initial advantage, but as of 1000 records, NoSQL database took the lead in execution speed, which was confirmed by 5000 records. With that amount of data, the hybrid was better placed than SQL. The advantages of NoSQL and Hybrid databases over SQL are noticeable and can be explained by the more flexible structure of their schemas. The additional time spent by the hybrid over the NoSQL DBMS can be explained by more advanced control of integrity rules, which NoSQL does not support.

The updates of 10, 100, 1000, and 5000 records took average 21, 131, 1342, and 11,482 ms on SQL database, 55, 92, 1276, and 10,890 on NoSQL, and 84, 140, 1385, and 12,210 on Hybrid, as shown in graph (Fig. 3b). Execution times, even at 5000 data, showed that all three databases achieved consistent results, with SQL database in this case slightly faster than the hybrid, while NoSQL database achieved the shortest times with 1000 and 5000 records. The main advantage of using NoSQL and hybrid over a SQL database represent their flexible structure, which can easily be changed. Should insert or update of data also involve modifying the record structure, it is expected that SQL would achieve noticeable slower execution time due to the need to execute additional ALTER command in order to change the structure.

Deleting of 10, 100, 1000, and 5000 records took 3, 6, 15, and 47 ms on average for SQL database, 31, 36, 43, and 76 for NoSQL database, and 68, 79, 81, and 145 for Hybrid, respectively. These results are shown on the graph (Fig. 3c). A graph with average runtime needed to display data from two database objects (two tables for SQL, two documents for NoSQL, and a table and a document for hybrid) is given in Fig. 3d. SQL databases handle this requirement by implementing an efficient JOIN statement, which achieves average execution times of 211, 1158, 11,735, and 27,286 ms over a set of 1000, 5000, 50,000, and 100,000 records, respectively. Hybrid SQL/NoSQL followed SQL's results closely with average times of 316, 1441, 13,141 and 29,951 ms. The explanation can be found in the efficiency of the introduced functionality of joining data from different hybrid components, as explained in the previous sections. Meanwhile, the lookup operator was used to connect data from different MongoDB documents. NoSQL database was noticeably slower, consuming 463, 2161, and 20,359 ms on average for 1000, 5000, and 50,000 records, respectively. For 100,000 records, the NoSQL database failed to execute the command, due to a limit on the allowed amount of aggregated data, which is a direct restriction of the lookup operator usage.

It can be seen from the above that NoSQL and Hybrid have the advantage of expensive data insert operations while updating (without changing the structure) and deleting are equally efficient with all three database representatives. SQL has proven to be the most efficient solution for joining large amounts of data from different tables. The inability to display 100,000 records confirmed the limitations of lookup operator and the NoSQL databases themselves to simultaneously read large amounts of data from different documents. Also, the tests confirmed the justification of the approach of splitting and executing "smaller" dedicated queries over the hybrid components and then integrating their results into SQL output. Usage of the architecture with new components has enabled the hybrid SQL/NoSQL database to closely follow the SQL database average execution time, even with the most demanding tests, while providing the NoSQL model flexibility, at the same time.

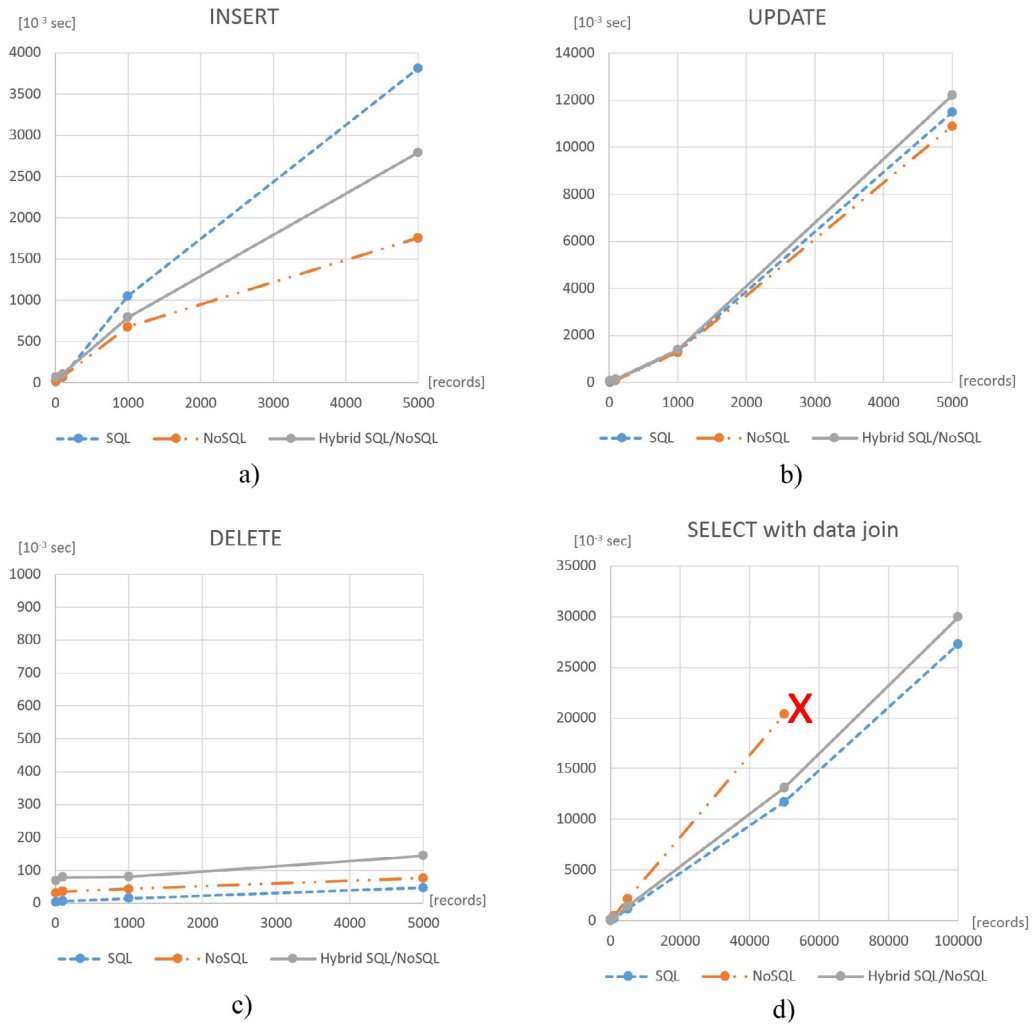

Fig. 3. Charts with average execution time for data insertion (a), data updating (b), data deleting (c), and data selecting (d).

6. Conclusion and future work

The problem of integration and uniform use of databases of different types in literature has been addressed in many ways. Even though they are in their early days, hybrid databases are a modern approach to tackling the described integration and usage challenges, which were recognized as the two most important NFRs for the new architecture. Numerous papers, which are mentioned in this work, prove that three-tier architecture is a good basis for the integration of databases of different types. Also, they opened the door for further research on complete integration and additional improvement of uniform use of different databases, especially when they represent components of hybrid databases.

The new approach, presented in this paper, enabled data from different hybrid components to be connected (by introducing support for data joining), as well as enabled centralization of hybrid SQL/NoSQL database administration. The presented architecture integrates SQL and NoSQL databases as components of a unique logical hybrid SQL/NoSQL database. By utilizing standardized SQL query language, it enables their universal use. The uniform use provides the user to enter exclusively SQL statements no matter if they shall be executed over the SQL or NoSQL component of hybrid. After analysis, decomposition and adjustment by the system, the entered query and its segments are executed over destination databases regardless of their type. New components of the architecture were introduced in order to provide that. This paper describes representative use cases, and the achieved effects for integration and uniform use of hybrid SQL/NoSQL database components. Use cases were tested on SQL, NoSQL and hybrid database (using new architecture).

The possibility of determining the number of hybrid components based on the design process might facilitate planning, developing and maintenance of the used architecture, which could be the direction of further research. Furthermore, one of the directions for development is introducing support for DDL (Data Description Language) statements, which would enable the creation of new objects using SQL language, no matter what type of object is going to be created (table, document, column etc.). That would open space for the development of a new component, which would be responsible for determining the destination component for the newly created object. Besides this, one of the directions for further development could represent support for adding more advanced SQL operations, which may have a potentially beneficial effect on performances. For example, MERGE and BULK INSERT are good candidates for reducing the number of trips from application to the database, which might influence the execution time. In combination with an ALTER statement it could potentially give advance to hybrids over SQL databases. Because tests showed that some operations like lookup do have limitations, optimization of statements is also an important direction for future research.

References

Abiteboul, S., Manolescu, I., Rigaux, P., Rousset, M.C., Senellart, P., 2011. Web Data Management. Cambridge University Press, New York, NY, USA.

Alomari, E., Barnawi, A., Sakr, S., 2015. Cdport: A portability framework for nosql datastores. Arab. J. Sci. Eng. 40 (9), 2531–2553. http://dx.doi.org/10.1007/s13369-015-1703-0.

Arenas, M., Diaz, G., Fokoue, A., Kementsietsidis, A., Srinivas, K., 2014. A principled approach to bridging the gap between graph data and their schemas. PVLDB 7 (8), 601–612.

Atzeni, P., Bugiotti, F., Rossi, L., 2012. Uniform access to non-relational database systems: The SOS platform. In: International Conference on Advanced Information Systems Engineering. Springer, Berlin, Germany, pp. 160–174.

Avison, D., Fitzgerald, G., 2003. Where now for development methodologies?. Commun. ACM 46 (1), 78–82. http://dx.doi.org/10.1145/602421.602423.

Bach, M., Werner, A., 2014. Standardization of nosql database languages. In: Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., Kostrzewa, D. (Eds.), Beyond Databases, Architectures, and Structures, Proceedings of 10th International Conference (BDAS 2014). Springer, Cham, Switzerland, pp. 50–60. http://dx.doi.org/10.1007/978-3-319-06932-6.

Badia, A., Lemire, D., 2011. A Call to arms: Revisiting database design. SIGMOD 40 (3), 61–69. http://dx.doi.org/10.1145/2070736.2070750.

Bjeladinovic, S., 2018. A fresh approach for hybrid SQL/NoSQL database design based on data structuredness. Enterpr. Inf. Syst. 12 (8–9), 1202–1220. http://dx.doi.org/10.1080/17517575.2018.1446102.

Buckler, C., 2016. Using JOINs in MongoDB NoSQL databases. Available on https://www.sitepoint.com/using-joins-in-mongodb-nosql-databases, (Retrieved: December 2019).

Bugiotti, F., Cabibbo, L., Atzeni, P., Torlone, R., 2014. Database design for NoSQL systems. In: International Conference on Conceptual Modeling. Springer International Publishing, pp. 223–231.

Curé, O., Hecht, R., Le Duc, C., Lamolle, M., 2011. Data integration over nosql stores using access path based mappings. In: International Conference on Database and Expert Systems Applications. Springer, Berlin, Germany, pp. 481–495.

Dasgupta, S., Coakley, K., Gupta, A., 2016. A semantic approach to polystores. In: 2016 IEEE International Conference on Big Data. IEEE, Washington, DC, USA, pp. 2555–2564. http://dx.doi.org/10.1109/BigData.2016.7840897.

Duggan, J., Elmore, A., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T., Zdonik, S., 2015. The bigdawg polystore system. ACM SIGMOD Rec. 44 (2), 11–16. http://dx.doi.org/10.1145/2814710.2814713.

Grolinger, K., Higashino, W., Tiwari, A., Capretz, M., 2013. Data management in cloud environments: NoSQL and NewSQL data stores. J. Cloud Comput.: Adv. Syst. Appl. 22 (2), http://dx.doi.org/10.1186/2192-113X-2-22.

Han, J., Haihong, E., Le, G., Du, J., 2011. Survey on NoSQL database. In: 6th International Conference on Pervasive Computing and Applications (ICPCA). IEEE, Washington, DC, USA, pp. 363–366. http://dx.doi.org/10.1109/ICPCA.2011.6106531.

Holzschuher, F., Peinl, R., 2013. Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops. ACM, New York, NY, USA, pp. 195–204.

Inmon, H.W., Strauss, D., Neushloss, G., 2008. DW 2.0 the Architecture for the Next Generation of Data Warehousing. Morgan Kaufman, San Francisco, CA.

Kharlamov, E., Mailis, T., Bereta, K., Bilidas, D., Brandt, S., Jimenez-Ruiz, E., Lamparter, S., Neuenstadt, C., Özçep, O., Soylu, A., Svingos, C., Xiao, G., Zheleznyakov, D., Calvanese, D., Horrocks, I., Giese, M., Ioannidis, Y., Kotidis, Y., Moller, R., Waaler, A., 2016. A semantic approach to polystores. In: 2016 IEEE International Conference on Big Data. IEEE, Washington, DC, USA, pp. 2565–2573. http://dx.doi.org/10.1109/BigData.2016.7840898.

Lake, P., Crowther, P., 2013. Concise Guide to Databases. Springer, London, UK.

Lawrence, R., 2014. Integration and virtualization of relational SQL and NoSQL Systems, including MySQL and MongoDB. In: International Conference on Computational Science and Computational Intelligence. IEEE, Washington, DC, USA, pp. 285–290.

Lebo, T., Del Rio, N., Fisher, P., Salisbury, C., 2017. A five-star rating scheme to assess application seamlessness. Semant. Web – Interoper. Usability Appl. IOS Press J. 8 (1), 43–63.

Maccioni, A., Basili, E., Torlone, R., 2016. QUEPA: Querying and exploring a polystore by augmentation. In: 2016 International Conference on Management of Data. Sigmod, San Francisco, California, USA, pp. 2133–2136. http://dx.doi.org/10.1145/2882903.2899393.

McHugh, J., Cuddihy, P.E., Williams, J.W., Aggour, K.S., Kumar, V.S., Mulwad, V., 2017. Integrated access to big data polystores through a knowledge-driven framework. In: 2017 IEEE International Conference on Big Data. IEEE, Boston, MA, USA, pp. 1494–1503. http://dx.doi.org/10.1109/BigData.2017.8258083.

Melton, J., Michels, J.E., Josifovski, V., Kulkarni, K., Schwarz, P., 2002. SQL/MED: a status report. ACM SIGMOD Rec. 31 (3), 81–89. http://dx.doi.org/10.1145/601858.601877.

Microsoft, 2019. Microsoft azure. Available on: https://azure.microsoft.com/en-us/, (Retrieved: January 2019).

MongoDB, 2019. The mongodb 3.2 manual. Available on: https://docs.mongodb.com/v3.2/ (Retrieved: January 2019).

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

Nickel, M., Tresp, V., 2013. An analysis of tensor models for learning on structured data. In: Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD 2013). Springer, Berlin, Germany, pp. 272–287. http://dx.doi.org/10.1007/978-3-642-40991-2_18.

Niu, N., Xu, L.D., Bi, Z., 2013. Enterprise information systems architecture – analysis and evaluation. IEEE Trans. Ind. Inform. 9 (4), 2147–2154.

Nyati, S.S., Pawar, S., Ingle, R., 2014. Performance evaluation of unstructured nosql data over distributed framework. In: International Conference on Advances in Computing, Communications and Informatics (ICACCI). IEEE, Washington, DC, USA, pp. 1623–1627.

van Ombergen, S., 2014. A Comparison of Five Document-Store Query Languages - Finding a Suitable Standard (Master thesis). University of Amsterdam, Amsterdam, Netherlands.

Oracle, 2019a. Oracle nosql database enterprise edition, version 18.1. [white paper]. Available on http://www.oracle.com/technetwork/products/nosqldb/learnmore/nosql-database-data-sheet-498054.pdf (Retrieved: January 2019).

Oracle, 2019b. Table expression. Available on: https://docs.oracle.com/javadb/10.6.2.1/ref/rreftableexpression.html#rreftableexpression (Retrieved: February 2019).

Polese, G., Vacca, M., 2009. A dialogue-based model for the query synchronization problem. In: IEEE 5th International Conference on Intelligent Computer Communication and Processing. IEEE, Washington, DC, USA, pp. 67–70.

Roijackers, J., Fletcher, G.H., 2013. On bridging relational and document-centric data stores. In: British National Conference on Databases. Springer, Berlin, Germany, pp. 135–148.

Sadalage, P.J., Fowler, M., 2012. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley, Boston, MA, USA.

Sellami, R., Bhiri, S., Defude, B., 2014. ODBAPI: a unified REST API for relational and NoSQL data stores. In: 2014 IEEE International Congress on Big Data. IEEE, Washington, DC, USA, pp. 653–660.

Shepelev, I.E., 2011. Identification of the hierarchical data structure. Pattern Recognit. Image Anal. 21 (2), 211–214.

da Silva, C., 2011. Data Modeling with NoSQL: How, when and Why? (Master thesis). Faculdade de Engenharia da Universidade do Porto, Porto, Portugal.

SolidIT, 2019. DB-engines ranking. Available on: https://db-engines.com/en/ranking (Retrieved: January 2019).

Sullivan, D., 2015. NoSQL for Mere Mortals. Addison-Wesley, Boston, MA, USA.

Tahara, D., Diamond, T., Abadi, D.J., 2014. Sinew: a SQL system for multi-structured data. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. ACM, New York, NY, USA, pp. 815–826.

Tatemura, J., Po, O., Hsiung, W.P., Hacigümüş, H., 2012. Partiqle: An elastic SQL engine over key–value stores. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, New York, NY, USA, pp. 629–632.

Valer, H., Sauer, C., Härder, T., 2013. XQuery processing over NoSQL stores. In: Proceedings of the 25th GI-Workshop Grundlagen von Datenbanken 2013, pp. 75-80.

Vilaça, R., Cruz, F., Pereira, J., Oliveira, R., 2013. An effective scalable SQL engine for NoSQL databases. In: IFIP International Conference on Distributed Applications and Interoperable Systems. Springer, Berlin, Germany, pp. 155–168.

Winand, M., 2017. What's new in SQL:2016. Available on: https://modern-sql.com/blog/2017-06/whats-new-in-sql-2016 (Retrieved: April 2018).

Zhu, J., Wang, A., 2012. Data Modeling for Big Data. [White Paper], CA technologies, Beijing, China.

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

Zoran Marjanovic is a Full Professor at Faculty of Organizational Sciences, University of Belgrade. His research interests are information systems development methodologies, databases, and semantic enterprise application interoperability.

Sladjan Babarogic is an Associate Professor at Faculty of Organizational Sciences, University of Belgrade. His research interests are information systems design, business process modelling, workflow and document management systems.