An integration approach of hybrid databases based on SQL in cloud computing environment

Changqing Li  |  Jianhua Gu

School of Computer Science and Engineering, Northwestern Polytechnical University, Xi'an, China

Correspondence
Changqing Li, School of Computer Science and Engineering, Northwestern Polytechnical University, Xi'an 710072, China.
Email: li_changqing@126.com

Summary
As the applications with big data in cloud computing environment grow, many existing systems expect to expand their service to support the dramatic increase of data, and modern software development for services computing and cloud computing software systems is no longer based on a single database but on existing multidatabases and this convergence needs new software architecture design. This paper proposes an integration approach to support hybrid database architecture, including MySQL, MongoDB, and Redis, to make it possible of allowing users to query data simultaneously from both relational SQL systems and NoSQL systems in a single SQL query. Two mechanisms are provided for constructing Redis’s indexes and semantic transforming between SQL and MongoDB API to add the SQL feature for these NoSQL databases. With the proposed approach, hybrid database systems can be performed in a flexible manner, ie, access can be either relational database or NoSQL, depending on the size of data. The approach can effectively reduce development complexity and improve development efficiency of the software systems with multidatabases. This is the result of further research on the related topic, which fills the gap ignored by relevant scholars in this field to make a little contribution to the further development of NoSQL technology.

KEYWORDS
big data, cloud computing, database services, hybrid databases, integration, NoSQL

1 | INTRODUCTION

1.1 | Background

Services computing, known as service-oriented computing, and cloud computing are two mainly coexisting paradigms that are widely demanded at present software systems, and most works focus on the problem of satisfying the one or multiple quality of service requirements for users while keeping satisfactory performance of the software systems. However, with the merging, sharing, and integration of different software systems, multidatabases have become the most important storage of current services computing and cloud computing software systems. Therefore, modern software development for services computing and cloud computing software systems is no longer based on a single database but on existing multidatabases and this convergence needs new software architecture and framework design. In particular, as the
applications of NoSQL (not only SQL) database systems become more and more popular, it also poses a greater challenge to the development methods and technical architecture of software systems.

A dramatic change, or called a revolution, is taking place in the field of database management technology as the growing of applications with big data in cloud computing environment. In June 1970, Edgar Frank Codd, a researcher of IBM company, published the famous paper, ie, *A Relational Model of Data for Large Shared Data Banks*, in Communications of ACM, which established the theoretical model of relational database. After decades of development, the relational database management system has gradually become mature, and the SQL (structured query language) used to access the relational database system also has been developed into the standard of all relational databases. However, big data and hybrid database system are becoming popular as cloud service blooms, and NoSQL database systems, such as *Redis*, *MongoDB*, *HBase*, and *Neo4J*, also have been developed rapidly.

NoSQL systems have been developed to support applications not well served by relational systems, often involving big data processing. Therefore, related research works have academic significance and practical values. Typical applications of NoSQL databases involve many important areas, such as the Internet, mobile computation, telecommunications, bioinformatics, education, and energy.

Major NoSQL databases can be classified into four categories, including key-value store, column-oriented store, document-oriented store, and graph store. Representative NoSQL database products include *Redis*, *Hbase*, *MongoDB*, and *Neo4j*, respectively, which belongs to one of the four types.

Key-value stores use the associative array, also known as a map or dictionary, as their fundamental data model. In this model, data is represented as a collection of key-value pairs. Document-oriented stores, in general, assume that documents encapsulate and encode data in some standard formats or encodings. Encodings in use include XML (eXtensible Markup Language), YAML (YAML Ain’t Markup Language), and JSON (JavaScript Object Notation) as well as binary forms like BSON (Binary JavaScript Object Notation). Documents are addressed in the database via a unique key that represents that document. Graph stores are designed for data whose relations are well represented as a graph consisting of elements interconnected with a finite number of relations between them. Column-oriented stores store data tables by column rather than by row, and the database can more precisely access the data it needs to answer a query rather than scanning and discarding unwanted data in rows.

Main advantages of NoSQL are the following aspects: reading and writing data quickly, supporting mass storage, easy to expand, and low cost. The NoSQL systems generally do not provide ACID (atomicity, consistency, isolation, and durability) transactional properties. By giving up ACID constraints, much higher performance and scalability can be achieved. Meanwhile, NoSQL also has some other inadequacies, such as does not support SQL, which is industry standard; lacking of reports; and other additional features.

Importantly, there are no common standard APIs for accessing the different NoSQL systems or standard query languages such as SQL. Typically, users must interact with these systems at a programming level with custom APIs. This reduces portability and requires system-specific code. In order to solve the problem, many scholars have used different techniques to build SQL access layer for some NoSQL databases. The usual practice of these researchers was building an intermediate layer that parses SQL queries and translates them to the query languages or APIs of specific NoSQL systems. Of course, due to the functional constraints of the NoSQL databases, usually, only a subset of SQL can be offered.

### 1.2 Motivation

It can be found that adding the SQL access layer is relatively easier for some NoSQL systems, such as *MongoDB* and *HBase*, because they have specific APIs, which can filter or sort row records according to multiple conditions. However, it is relatively difficult to provide the SQL support for another some NoSQL data stores, such as *Redis* and *Memcached*, because they do not have corresponding APIs; therefore, it is necessary constructing such APIs for them firstly.

While NoSQL database has the ability to manage big data, relational database still has superiority with middle or small scale of data, and most of the existing systems are based on relational database, but with the growth of data size, enterprise tends to handle big data with NoSQL database for analysis or wants to get faster access on big data. In addition, how to query simultaneously from multiple hybrid data-sources when integrating these heterogeneous databases? This also is a difficult problem, which involves the parsing, optimization, and routing calculation of SQL statement and also requires the merging, grouping, and sorting for the results returned by each data source.

Unfortunately, as far as the authors know, there is little research on the aforementioned problems in the existing literary works.
Therefore, by the way of SQL, more efforts should be made to integrate heterogeneous data sources including MySQL, MongoDB, and Redis, which mainly involves the following aspects:

1. SQL parsing, optimization, and routing calculation;
2. Semantic transformation between SQL and MongoDB API. In order to provide SQL feature for MongoDB, it is necessary to implement the semantic transformation function from SQL to MongoDB;
3. Establishment and maintenance of auxiliary indexes of Redis data-store. The indexes are built when the row data is inserted into the data store, and corresponding maintenance works of them are performed when the row data is modified or removed;
4. Querying from multiple hybrid data-sources. This also is a difficult problem;
5. The merging, grouping, and sorting of the results returned by each data source.

1.3 Our contributions

This paper proposes an integration approach that can effectively reduce the development complexity and improve the development efficiency of the software systems with multidatabases. The key algorithms were given in the form of pseudo program code, and the approach was evaluated, including the qualitative comparison and the quantitative analysis. This is the result of further research on the related topic, which fills the gap ignored by relevant scholars in this field to make a little contribution to the further development of the NoSQL technology.

Our contributions are summarized as follows.
1. This paper presents an integration approach to make it possible of allowing users to query data from both relational SQL systems and NoSQL systems, including MySQL, MongoDB, and Redis, in a single SQL query.
2. With the proposed approach, two mechanisms are provided for constructing Redis’s indexes and semantic transforming between SQL and MongoDB API.
3. With the proposed approach, hybrid database systems can be performed in a flexible manner, ie, access can be either relational databases or NoSQL data-stores, depending on the size of data.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the related works. Section 3 describes the transformation method of nested queries. Section 4 describes the model’s architecture and application patterns. Section 5 describes the semantic transformation method between SQL and MongoDB API, the structures of the two kinds of auxiliary indexes of Redis data-store, and the querying method from multiple hybrid data-sources. Section 6 is the approach evaluation. Section 7 concludes this paper.

2 RELATED WORKS

Attempting to provide an SQL access layer on top of selected NoSQL database, a lot of efforts have been made by many scholars\textsuperscript{13,23-32} to increase the portability of the software system.

Some scholars\textsuperscript{23-25} studied the implementation method of SQL access layer of HBase database system. In literature,\textsuperscript{23} Apache Phoenix was used, which is an open-source SQL translator for HBase, and it allows database users who are familiar with SQL to access HBase with frequently used SQL commands. The authors in the literature\textsuperscript{24} structured a full-edged SQL engine, which can be integrated atop of HBase NoSQL database. Tatemura et al\textsuperscript{25} also developed an SQL engine, called Partiqle, that runs on top of the HBase store, which is an open-source NoSQL database.

Other researchers\textsuperscript{13,26,27} explored the implementation method of the SQL access layer of MongoDB database systems. Khan and Mane\textsuperscript{26} proposed a method that provides SQL query language support to the NoSQL database MongoDB by appending an interface (Metadata) between the application layer and database layer. The writers in literature\textsuperscript{27} identified the subset of SQL for access to NoSQL systems and developed a middleware that can translate SQL queries to the query languages of the connected NoSQL systems, including Cassandra and MongoDB. In literature,\textsuperscript{13} a virtualization system is built on top of the NoSQL sources that can translate SQL queries into the source-specific APIs. The virtualization architecture allows users to query and join data from both NoSQL (with specific focus on MongoDB) and relational SQL systems (with special focus on MySQL) in a single SQL query.

Some people\textsuperscript{27,28} tried to implement the SQL access layer of other NoSQL database systems, including SimpleDB and Cassandra. The literature\textsuperscript{28} introduced SimpleSQL, a relational layer over Amazon SimpleDB, one of the most popular document-oriented NoSQL databases. SimpleSQL offers an SQL interface that accesses to SimpleDB and supports the four traditional manipulation operations, ie, select, update, insert, and delete.
At the same time, in some literary works, the researchers described generally how to add SQL feature to NoSQL database systems, without specifying the specific NoSQL names. The authors in the literature presented a unified relational view concept, which supports SQL queries, based on virtual schema on top of SQL world and NoSQL world, which integrates the idea of “Schema-less for Write” from NoSQL world and “Schema-Rich for declarative Query” from SQL world through management of JSON data in a single RDBMS product without the need of polyglot persistence. The literature described the design of a layer above a traditional relational database system that enables standard SQL-compliant queries to be issued over multistructured data (relational, NoSQL, or other types of semistructured data). This is a hybrid storage solution, which has a query rewriter that modifies queries before sending them to the storage layer for execution. Zhang et al built an SQL engine that converts standard SQL requests into various types of access to the NoSQL database, which acts as wrapper integrated into NoSQL clusters each nodes. The authors in the literature developed a software layer, called API/Language support (SQL, JDBC), which was used to process SQL queries, and the queries were forwarded by a distributed query processor to a specific NoSQL store or a relational database.

In addition, it should be noted that some researchers also proposed some common application program interfaces by analyzing the programming interfaces of different database management systems to increase the portability of software systems. For example, literature abstracted a set of common programming interfaces for different NoSQL database systems including Redis, MongoDB, and HBase, called SOS (Save Our Systems). Focusing on NoSQL systems such as CouchDB, MongoDB, and Riak, literature also defined a set of generic programming interfaces, called ODBAPI (OPEN-PaaS-DataBase API). Since these programming interfaces are not the standard languages in the industry, therefore, they are not stable. At the same time, it takes valuable time for software developers to familiarize with these programming interfaces.

To sum up, in order to solve the problem of the lack of SQL features of NoSQL databases, many scholars have studied various techniques to construct an SQL access layer for specific NoSQL store. This SQL intermediate layer can transform SQL queries into a specific query language or APIs of the selected NoSQL database.

At the same time, how to integrate relational databases and NoSQL data-stores based on SQL? Some scholars also finished some preliminary exploration on the problem. However, the related research works have the following shortcomings.

1. Lack of research on SQL implementation method of NoSQL databases belonging to the key-value type. The solution was not given in the aforementioned literary works about how to provide SQL support for the key-value NoSQL database like the famous Redis system. As we all know, this kind of NoSQL data stores are not like other NoSQL databases, such as MongoDB and HBase. Since they do not have corresponding APIs, which can filter or sort row records according to multiple conditions, therefore, it is necessary to construct such APIs for them firstly. In other words, much more efforts must be made, providing SQL support for such NoSQL systems.

2. Further research works need to be carried out on the integration method of relational databases and NoSQL data-stores. How to query simultaneously from multiple hybrid data-sources when integrating these heterogeneous databases? This also is a difficult problem, which involves the parsing, optimization, and routing calculation of SQL statement and also requires the merging, grouping, and sorting for the results returned by each data source.

Importantly, as everyone knows that, in a single SQL statement, the typical mechanisms for accessing to multiple tables include nested predicate and join predicate. Therefore, in order to integrate heterogeneous database systems, it is necessary to solve the problems including arbitrary complex nested query and join. Firstly, literary works did not give a solution to solve arbitrary complex nested query problem. Secondly, in solving the join problem, literary works did not take into account the situation of specific NoSQL database (for example, the left outer join of MongoDB).

3 | TRANSFORMATION OF NESTED QUERIES

Nested query is a powerful expression mechanism in SQL. It not only brings convenience to the application but also brings great challenges to query optimization.

The research paper of Kim in this field has become a classical literature because it classified the typical nested queries reasonably and put forward the corresponding transformation algorithm.

Considering whether a subquery has join condition and aggregate function, Kim defined the following five basic forms of nested queries.

Type-A: the inner query block has no reference to the table of the outer query block, and the query result is an aggregate function.
Type-N: the inner query block has no reference to the table of the outer query block, and the query result has no aggregation function.

Type-J: the inner query block has a reference to the table of the outer query block, and the query result has no aggregation function.

Type-JA: the inner query block has a reference to the table of the outer query block, and the query result set has an aggregate function.

Type-D: in a query formed by a join predicate and a division predicate, there are two inner query blocks and at least one join predicate references, i.e., the outer query block.

Since Type-D is less used in real database systems, only four other nested query transformation algorithms were given by Kim. Finally, a general algorithm is formed to deal with arbitrary complex nested queries, which is called nonnesting of nested query.

However, Kim’s algorithms were quickly found to have minor bugs in some particular scenarios, and the bugs needed to be fixed. Ganski and Wong solved the defects in Kim’s NEST-JA algorithm and extended it to commonly used clauses in SQL, including EXISTS, NOT EXISTS, ANY, and ALL. The improved algorithm by Ganski et al is called the Nest-G algorithm (see Algorithm 1).

Algorithm 1 Nest-G

```
input: Query_Block: is a pointer to a SQL query block, and possibly with descendant inner query blocks nested within it.
output: Rt: the transformed SQL query block.

for (each predicate in the Where clause of Query_Block) do
    if (predicate is a nested predicate) then
        /* Recursively invokes the algorithm: */
        Nest-G(Inner_Query_Block);
        /* Determines type of nesting and calls appropriate transformation algorithm: */
        if (Select clause of Inner_Query_Block contains aggregate function) then
            /* Nesting contains join predicate referencing a relation which is not in its From clause: */
            Nest-JA2(Inner_Query_Block);
            Nest-N-J(Query_Block.Inner_Query_Block);
        else
            /* Nesting is type-A: */
            Nest-A(Inner_Query_Block);
        else
            /* Nesting is type-N or type-J: */
            Nest-N-J(Query_Block.Inner_Query_Block);

return Rt;
```

Generally, the typical mechanisms for accessing multiple tables in a single SQL statement are join predicate and nested predicate; therefore, the nonnesting of nested query is to transform a query in the form of nested predicate into a join predicate. The Nest-G algorithm accomplishes this function essentially.

The core idea of the Nest-G algorithm is transforming a query in the form of nested predicate into a join predicate. The details are explained as follows.

1. In lines 1 to 17, each predicate in the where clause of a query block is processed, and from lines 2, the Nest-G algorithm begins its real transformation when a predicate is a nested predicate.
2. In line 4, the Nest-G algorithm is invoked recursively.
3. In lines 6 to 16, the Nest-G algorithm determines the type of nesting and calls appropriate transformation algorithm.

The following details are made. (1) Lines 9 and 10 transform type-JA; it is transformed firstly into a nested query of type-N or type-J type by Nest-JA2 algorithm, and then it is transformed into a nonnested form by Nest-N-J algorithm. (2) Line 13 transforms type-A. (3) In line 16, type-N or type-J is transformed.
The time complexity of the Nest-G algorithm is analyzed as follows.
1. The Nest-G algorithm is invoked recursively, which can be done within \(O(|D|)\) time (Line 4), and \(D\) represents the nested depth.
2. For all branches of the algorithm (Lines 6 to 16), only the worst scenario needs to be considered. When nesting is type-JA (Lines 9 and 10), type-N or type-J (Line 16), the join operation should be performed, which can be done within \(O(|M||N|)\) time, where \(M\) and \(N\) represent the number of rows actually participating in the join operation.

Thus, the time complexity of the Nest-G algorithm is \(O(|D||M||N|)\).

The two transformation scenarios for the Nest-G algorithm are as follows.
1. For type-A transformation, subquery blocks need to be executed recursively from inner block to outer block.
2. The join predicate was produced in the transformed results of other types of nested queries.

To sum up, the Nest-G algorithm can transform arbitrary complex nested query into equivalent nonnested query and join predicates are often included in the transformed query statement. Therefore, in the environment of hybrid database systems, including MySQL, MongoDB, and Redis, it is necessary to implement a join algorithm over cross-database systems to use the Nest-G algorithm because it is different from the application scenario of a single relational database system.

## 4 MSI’s Architecture and Application Patterns

### 4.1 MSI’s Architecture

Modern software development for services computing and cloud computing software systems is no longer based on a single database but on existing multidatabases and this convergence needs new software architecture and framework design. The integration model of relational databases and NoSQL data-stores proposed in this paper is shown in Figure 1, called MSI (multiple sources integration), which is based on the underlying layers of the operating system and the multiple hybrid database management systems. The SQL statements are the MSI’s input and the result of the SQL process is the MSI’s output.

In the overall aspects, MSI mainly includes eight components, ie, API dispatcher component, SQL parser component, SQL optimizer component, SQL router component, SQL executor component, DBMS (data base management system) adapter component, result merger component, and meta-data management component.

1. API dispatcher component

The API dispatcher component mainly provides calling methods of SQL APIs for users. When the user inputs the SQL statements, the API dispatcher component firstly translates the input SQL statements into the specific computer language.
objects by the SQL parser component. Then, the specific computer language objects are transformed into the developed APIs by the DBMS adapter component. Finally, the execution results are returned by calling the native APIs, which also lies in the DBMS adapter component and provided by the selected DBMS driver.

2. SQL parser component

The SQL parser component mainly includes lexical analysis, syntax analysis, SQL analysis, tree traveler, objects encapsulation, and error handling.

In the prototype system of this paper’s presented model, the authors used the ANTLR, which is an open source tool, to implement the SQL parser. In accordance with the grammars of ANTLR, the SQL specification was converted into the SQL syntax file of the ANTLR tool. Then, ANTLR used the finished SQL syntax file to generate the corresponding lexical analyzer and syntax analyzer.

The input of lexical analyzer is the SQL string, and its output is the phrase types, which is called tokens by the ANTLR. The input SQL string is converted into abstract syntax tree by the grammar parser using these tokens and syntax rules. Of course, the grammar parser can produce some things, such as the auxiliary data structures, character table, and the flow chart data.

The SQL analysis module can obtain the various parts of the SQL string by calling the abstract syntax tree traveler and can store the results into the auxiliary data structures. By accessing to the auxiliary data structures generated by the SQL analysis module, the objects encapsulation module can convert the SQL statements into the language objects, which are described by the specific computer language. For example, it can convert the select statement, which is a kind of SQL statements, into the Java language objects.

3. SQL optimizer component

Normally, the SQL statement needs to be optimized after parsing. For example, for the where clause, it is usually required to deal with complex combinatorial query conditions with parentheses in practical applications.

Therefore, 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. This situation is beyond the scope of this paper, and it is planned to discuss in another paper.

In addition, for arbitrary complex nested query statements, it also needs to be transformed according to the Nest-G algorithm, which was described in the Section 3 of this paper.

4. SQL router component

The SQL router component is responsible for distributing the SQL statement to the correct database management system. In particular, when an SQL statement requires access to different databases, the SQL statement must be decomposed into several parts so that the specific database only receives and processes related part.

The SQL router component obtains the configuration information of each database by accessing to the meta-data management component. In addition, the component is most closely associated with the SQL parser component and the SQL optimizer component.

5. SQL executor component

The SQL executor component can distribute the decomposed SQL statement to the correct database management system by interoperating with the DBMS adapter component. After that, the SQL executor component receives the result returned from specific database management system. Finally, the SQL executor component sends the final data to the result merger component for further processing.

6. DBMS adapter component

The DBMS adapter component is mainly composed of semantic checking, parameters generation, APIs calling, run-time supporter, and error handling. The inputs of DBMS adapter component are the language objects generated by the SQL parser component, which has probably been optimized by the SQL optimizer component, and its outputs are the calls of the encapsulated APIs for the selected NoSQL database.

The semantic checking module is mainly responsible for checking the language objects produced by the upper layer. For instance, the “like” keyword cannot be used in the where clause when the field is not a string type.

The parameters generation module is mainly used to extract the corresponding parameters of each SQL statement and translates them into specific parameter objects, which will be used in the encapsulated APIs for the NoSQL data-store. For example, when dealing with the language objects, which are formed by a select statement, it is necessary to process these parameter objects related to the from clause, the where clause, the order by clause, the limit clause, and so on.

According to the differences of the SQL statements, APIs calling module, respectively, calls different encapsulated APIs for the selected NoSQL database.
Run-time supporter mainly includes attributes configuration, database connection pool, ANTLR run-time library, the encapsulated APIs, and the driver of the selected NoSQL database. It provides supports for the upper layer and interacts with the NoSQL data-store management system by the driver.

7. Result merger component
Since data is stored in different databases, it is necessary to merge and sort the results returned by these hybrid database management systems, and data aggregation is mainly aimed at group by operation. In addition, this function is frequently used in data query statements.

Result merger component uses the heap sorting algorithm for merging and sorting the ordered data, and then returns the processed data to the client.

8. Meta-data management component
Meta-data is information about the organization of data, data fields, and their relationships. In short, meta-data is the data about data. Here, the meta-data management component is responsible for managing the various configuration parameters of the database and the results of the SQL statement having been parsed, optimized, and routed.

For example, two tables in a relational database can be used to define the structure of all business tables, as shown in Figure 2, these tables is used to support the system’s meta-data. Some necessary comments about the sample figure are as follows.

a. As shown in the upper part of Figure 2, the basic information of a business table is defined in the main table called `TablesBasicInfo`, whereas all fields of the specific business table are defined in the subtable called `FieldsBasicInfo`, and the two meta-data tables are associated by the field called `TableID`.

b. As shown in the middle section of Figure 2, three business tables, including `Type`, `Customer`, and `Order`, are defined in the main meta-data table called `TablesBasicInfo`, and their real row records are stored, respectively, in the Redis, MongoDB, and MySQL physical databases.

c. As shown in the lower part of Figure 2, all fields of each business table are defined in the meta-data table called `FieldsBasicInfo`. For instance, the three fields of the business table `Type` are demonstrated in the figure. Note that the value of the `FieldType` is that `P` represents the primary key, `F` represents the foreign key, and `C` represents the common field. At the same time, it also needs to fill the identification of associated business table in the `RelatedTableID` field when `FieldType` is `F`.

### 4.2 Application patterns

The application patterns of MSI mainly include the following three types.
1. **OLTP pattern**

As shown in Figure 3, MSI is mainly used as a support tool for OLTP (on-line transaction processing) application. For example, it can be applied in such scenarios as e-commerce, e-government, internet of things, and smart cities. In these scenarios, multiple database systems are often involved, including relational databases and NoSQL data-stores.

2. **Big data analysis pattern**

The MSI can work in two places, as shown in Figure 4. On the one hand, it can be used as a support tool for big data analysis application, accessing data warehouse system, such as HBase NoSQL database, by SQL. On the other hand, it can be applied in data synchronizer, which transfers data from the OLTP database system to data warehouse system.

3. **Hybrid pattern**

Hybrid pattern, as shown in Figure 5, is a coexist way of the OLTP pattern and the big data analysis pattern, which is also the most common style in practical applications. On the one hand, MSI acts as a bridge between application layer...
(involving OLTP application and big data analysis application) and data storage layer (including OLTP database systems and data warehouse system). On the other hand, it can be applied among different data storage systems.

5 | INTEGRATION APPROACH

The integration approach of relational databases and NoSQL data-stores based on SQL, including MySQL, MongoDB, and Redis, involves mainly three aspects, ie, semantic transformation between SQL and MongoDB API, construction methods of Redis’s indexes, and querying from multiple hybrid data sources.

5.1 | Semantic transformation between SQL and MongoDB API

In order to provide SQL feature for MongoDB, it is necessary to implement the semantic transformation function from SQL to MongoDB.

For SQL and MongoDB, there are differences in terms and concepts between them, and they are not exactly the same in data aggregation operations. The comparison between SQL and MongoDB’s shell language is shown in Figure 6, and these operations in SQL language are equivalent to the CRUD (create, retrieve, update, and delete). As can be seen from the figure, the difference lies in the grammatical format. For example, the SQL syntax for inserting a row record is `INSERT INTO {Table Name} {Field Name List} VALUES({Field Value List})`, whereas, in the MongoDB database, its shell language syntax for the same operation is “db.” + `{Table Name}` + “.insert( { Field Name : Field Value }) , ...” .

Here, a typical query operation is taken as an example to describe the transformation method from SQL to the MongoDB’s Java language API. The SQL example is shown in Figure 7.

The main transformation steps are as follows.

1. SQL parsing

The `select` statement is parsed by the SQL parsing component to produce a Java language object called “selectStatement”, as shown in Figure 7, and which has different properties that correspond to the various parts of the `select` statement.

   - In the `tableName` property, the table name of the database is stored.
   - In the `displayFields` property, the list of field names to display is stored as an array.
   - In the `sortFields` property, the field information is stored, which is used to sort the row records. It is stored in the form of an array, and the elements of the array are sort objects, which mainly include the field name and sort type.
   - In the `limit` property, the constraint parameter is stored for the query result.

   The processing of the `where` clause is more complicated. A single condition is encapsulated as an object, like “E-1”, stored in the `conditionMap` property, and in the `conditionExp` property, each query condition is replaced by its ID, for instance, “e1” represents “Course = English”. In addition, the relation “or” among conditions is replaced by “+” symbol,

*FIGURE 6*  The comparison between structured query language and MongoDB’s shell language
and the relation “and” is replaced by a space symbol. In this way, the entire where clause becomes \((e1+e2)(e3+e4)\), which is mainly taken for optimization purpose.

2. Optimization of query conditions

This task is carried out by the SQL optimization component, which mainly involves the expansion of logical expressions and the simplification of expressions based on the *Espresso* algorithm. According to the distributive law, the expression of \((e1+e2)(e3+e4)\) is transformed into the form of \(e1 \cdot e3 + e1 \cdot e4 + e2 \cdot e3 + e2 \cdot e4\).

Finally, \(\text{conditionsList}\), the other property of the \(\text{selectStatement}\) query object, is produced, as shown in Figure 8. As can be seen from the figure, two kinds of arrays are used to store the final logical expressions. In the first kind of array, \(\text{CL}\) array, the relation among the elements is “or”, and in the second kind of array, such as \(A\), \(B\), \(C\), and \(D\) array, the relation among the elements is “and”.

3. Transforming of clauses

The transformation of SQL clauses is transforming every part of SQL statement into equivalent statements of MongoDB API based on the relevant parameters obtained in the previous steps.

For example, based on the \(\text{displayFields}\) property of \(\text{selectStatement}\) object, the converted equivalent statements are as follows:

\[
\text{List < String > theSelectFieldsList = selectStatement.getSelectFields();}
\]

\[
\text{Bson fieldsList = Projections.fields( Projections.include( theSelectFieldsList ), Projections.excludeId() ).}
\]

In addition, according to the \(\text{conditionsList}\) property of \(\text{selectStatement}\) object, the \(\text{where}\) clause can be easily transformed into the statements of MongoDB API by means of the nested loop.

In general, SQL statement is very simple, whereas equivalent MongoDB API statements are very verbose. For example, the number of MongoDB API statements, which are equivalent to this example, is dozens of lines. Therefore, due to the length of this article, the full program codes are not given here.
5.2 Construction methods of Redis's indexes

In order to make Redis supporting the SQL feature, the most important task is to construct two types of data indexes for it, including the primary index and the secondary index. The goal of the primary index and the secondary index is to support the mapping of SQL into the Redis objects. In addition, in order to clearly describe the construction process of these two kinds of indexes, a simplified example is demonstrated here. Table 1 is a database table named Student, which has three fields, ie, RowId, Name, and Age, where RowId is the primary key of the table.

1. Construction method of primary index

The primary index is constructed as follows.

a. Storing row records into map data structures

As shown in Figure 9, the three row records in the sample database table are stored in three map data structures, and the map objects are respectively named Map-1, Map-2, and Map-3. At the same time, in the map data structure, “key” holds the “field name” of the row record, whereas the “field value” of the row record is stored into the “value” part.

b. Saving the map data objects into a hash table

As shown in Figure 9, the map objects where the row records reside are assembled into a hash table. In the “key” of the hash table, the primary index identifier of the row record is saved, whereas the identification of the map object in which the row record is located is saved into the “value” part of the hash table. It is important to note that the primary index identifier of a row record is constructed according to the following rule:

“Table.” + {TableName} + “.” + {RowId}.

For example, the Map-1 object can be quickly queried in the hash table by the key, ie, “Table.Student.20”, and all field values also can be gotten finally.

2. Construction method of secondary index

The secondary index is constructed as follows.

a. Storing all the field values of each field into the B+ tree data structures, respectively

As everyone knows, a B+ tree is a type of tree data structure, which was proposed firstly in the literature. At the same time, a B+ tree is a dynamic and multilevel index, which has maximum and minimum bounds on the number of keys in each tree node.

| RowId | Name | Age |
|-------|------|-----|
| 20    | Jim  | 18  |
| 30    | Tom  | 22  |
| 50    | Rose | 19  |
As shown in Figure 10, for the “Name” and “Age” fields in the sample database table, the corresponding B+ tree objects are created, where all the field values of each field are saved. At the same time, the leaf nodes of each B+ tree refer to the primary key of the table as the data part, and the same field value can have different primary keys.

**b. Saving the B+ tree objects into another hash table**

As shown in Figure 10, all B+ tree data objects are assembled into another hash table. In the “key” of the hash table, the secondary index identifier of the B+ tree object is saved, whereas the identification of the B+ tree object is saved into the “value” part of the hash table. It is important to note that the secondary index identifier of the B+ tree object is constructed according to the following rule:

“Index.Table.” + {Table name} + “.” + {Field Name}.

For example, the “B+Tree-1” object can be quickly found by the key, i.e., “Index.Table.Student.Name”, in the hash table, and the field values also can be gotten, which make up the B+ tree. Of course, all primary keys of the row records, which match the query conditions, also can be obtained.

Based on the two kinds of data indexes, it is easy to construct the APIs accessing to the rows of Redis database.

---

**5.3 | Querying from multiple hybrid data sources**

In a single SQL statement, the typical mechanisms for accessing to multiple tables include nested predicate and join predicate, as shown in the third section of this paper, and the general Nest-G algorithm can transform arbitrary complex nested query into an equivalent nonnested form, which usually contains the join predicate.

Therefore, in the hybrid environment of three heterogeneous databases including MySQL, MongoDB, and Redis, how to implement the join operation crossing multiple hybrid data-sources becomes the key. In the following parts, the authors intend to describe the basic join algorithm firstly, and then give an example to describe the joint query method from heterogeneous data sources.

**1. Basic join algorithm**

In general, join operation is not well supported in the NoSQL databases. For example, MongoDB only supports *left outer* join, whereas Redis does not support join operation at all. In the hybrid environment of three heterogeneous databases, including MySQL, MongoDB, and Redis, the basic join method between two tables is shown in Algorithm 2.
The core idea of the BasicJoin algorithm is performing join operation between two tables, which could come from MySQL, MongoDB, or Redis. The details are explained as follows.

**Algorithm 2 BasicJoin**

**input**: TL, TR: the tables on which the join operation need be performed, SQL: it includes the join operation, Sp: Size of page, namely the numbers of rows per page in a paging query.

**output**: Rt: the result of SQL

1. \textbf{if} (TL is MySQL and (TR is MySQL)) \textbf{then}
   
   \[ Rt \leftarrow \text{Push-down join to MySQL}; \]

2. \textbf{else}

   3. \textbf{if} (only TL is MySQL and (TR is a small table)) \textbf{then}
      
      \[ T1 \leftarrow \text{Create a temporary table in MySQL}; \]
      
      \[ T1 \leftarrow \text{Query from TR}; \text{Rewrite the SQL}; \]
      
      \[ Rt \leftarrow \text{Push-down join to MySQL}; \]

   4. \textbf{else}

      5. \textbf{if} (TL and TR is MongoDB and (Left Outer Join)) \textbf{then}

      6. \textbf{else} (exists filter in TL and (exists filter in TR)) \textbf{then}

      7. \[ RowSet \leftarrow L, RowSet \leftarrow R \leftarrow \text{Parallel filter on TL and TR}; \]

   8. \textbf{else}

      9. \textbf{if} (only exists filter in TL) \textbf{then}

      10. \[ RowSet \leftarrow L \leftarrow \text{filter on TL}; RowSet \leftarrow R \leftarrow \text{TR}; \]

   11. \textbf{else}

      12. \[ RowSet \leftarrow R \leftarrow \text{filter on TR}; RowSet \leftarrow L \leftarrow \text{TL}; \]

   13. \textbf{else}

      14. \[ RowSet \leftarrow L \leftarrow \text{TL}; RowSet \leftarrow R \leftarrow \text{TR}; \]

15. \textbf{for} (each page Pi of RowSet-L) \textbf{do}

16. \textbf{if} (Pi not in memory) \textbf{then}

17. \[ \text{Read page Pi into memory}; \]

18. \textbf{for} (each row Ri in page Pi) \textbf{do}

19. \textbf{for} (each page Pj of RowSet-R) \textbf{do}

20. \textbf{if} (Pj not in memory) \textbf{then}

21. \[ \text{Read page Pj into memory}; \]

22. \textbf{for} (each row Rj in page Pj) \textbf{do}

23. \textbf{if} (Ri,Rj meet join condition) \textbf{then}

24. \[ Rt \leftarrow \text{Join Ri, Rj}; \]

25. \textbf{return} Rt;

a. In lines 1 and 2, if the join operation needs performing between two tables of MySQL, the join operation will be pushed down directly into MySQL.

b. In lines 4 to 7, if the join operation needs finishing between MySQL and an another small table, which could possibly be MongoDB or Redis, the data stored in the small table is passed into the temporary table after creating a temporary table in MySQL, and the rewritten SQL statement is pushed down into MySQL to perform the join action finally.
c. In lines 9 to 11, if the left outer join needs completing between two tables of MongoDB, then the SQL statement will be transformed into the style of MongoDB API, and the join task is pushed down into MongoDB finally.

d. In lines 13 to 32, in addition to the aforementioned three scenarios, the filter action needs performing firstly before finishing the join task, then the basic join operation will be done in the way of the nested loop. In particular, if there are filtering conditions in both tables, the algorithm can perform data filtering operations in the way of parallel computing. At the same time, it must be pointed out that using SQL to query large amounts of data from databases, including relational databases, can lead to memory overflows. In order to avoid this situation, the usual practice is paging queries from a database, and the SQL language also provides technical support for this, for example, “Select * from tableName limit 1000 offset 2000”, where “limit 1000” limits the number of records returned by the database system each time and “offset 2000” indicates the number of records offset from the result set. Therefore, parameter “Sp”, size of page, can indicate the numbers of rows per page in a paging query.

The time complexity of the BasicJoin algorithm is analyzed as follows.

For all branches of the algorithm (Lines 1 to 32), only the worst scenario (Lines 23 to 32) needs to be considered, in which, the join operation can be done within $O(|M| \times |N|)$ time, where $M$ and $N$ represent the number of rows actually participating in the join operation.

Thus, the time complexity of the BasicJoin algorithm is $O(|M| \times |N|)$.

2. Typical example

How to query simultaneously from multiple hybrid data-sources? Here, the implementation method can be illustrated by a fictional typical example.

As shown in Figure 11, assume that there are three tables, including type, customer, and order, which were stored in Redis, MongoDB, and MySQL, respectively.

Now, all orders of the customer, whose type is “A” and who has the largest customer ID, need to be queried from these three hybrid databases, and the nested SQL statement is as follows.

```sql
SELECT oid, odate, oprice
FROM Order
WHERE cid = (SELECT MAX(cid)
              FROM Customer
              WHERE tid IN (SELECT tid
                             FROM Type
                             WHERE tcode = ‘A’))
```

This is a nested query involving three tables, from inner query block to outer query block, and it includes a type-N nested query and a type-A nested query. The query can be transformed recursively according to the Nest-G algorithm (see Algorithm 1) and the BasicJoin algorithm (see Algorithm 2). The steps are as follows.

Firstly, the inner nested query block can be transformed according to the Nest-N-J subalgorithm in the Nest-G algorithm. The result is as follows:

```sql
SELECT oid, odate, oprice
FROM Order
WHERE cid = (SELECT MAX(cid)
              FROM Customer, Type
              WHERE Customer.tid = Type.tid AND Type.tcode = ‘A’)
```
Secondly, from the query statement obtained from the previous step, it can be seen that the current nested query is type-A nested query. Therefore, the current query can be transformed according to the Nest-A subalgorithm in the Nest-G algorithm. In other words, the inner layer query block can be processed firstly according to the BasicJoin algorithm, as it involves a join operation between two tables, and then the result, assumed as $X$, will be used in the aforementioned statement to replace the inner layer query block. After this step, the original query statement becomes as follows:

```
SELECT oid, odate, oprice
FROM Order
WHERE cid = X
```

Thirdly, the outermost query block can be executed. That is, it will be pushed down into MySQL to get the final query result.

6 | APPROACH EVALUATION

The integration approach proposed in this paper is evaluated in this section. First of all, the qualitative comparison between the MSI model proposed in this paper and the Unity model is carried out. Secondly, the performance of the initial prototype is quantitatively analyzed. Finally, the evaluation results are summarized.

6.1 | Qualitative comparison

In this section, the MSI model proposed in this study is qualitatively compared with the Unity model, which was proposed in the literature. The Unity model is chosen because it has some similar components. As shown in Table 2, qualitative comparison covers 10 main features, including the language adopted for integration, parser of language, and other aspects.

As can be seen from Table 2, the two models are similar in the following points.
1. Both use SQL as the language to access multiple hybrid database systems, which is mainly due to the fact that SQL is the industry standard, which can reduce the learning difficulty of software developers and improve the development efficiency of the system.
2. Both can access multiple hybrid data sources simultaneously, this also is the common goal of the two models. At the same time, there are significant differences between the two models in the following aspects.
1. They are different in the way of parsing SQL. In the Unity model, the SQL parser was implemented based on JavaCC, which is a generator that can generate lexical analyzer and grammatical parser, whereas, in the MSI model, the SQL parser was designed based on ANTLR, which also is an open source generator that can produce compiler. By contrast, the latter is more flexible, but more efforts are needed.
2. The ability to access Redis is different. The MSI model supports accessing to Redis, which is a NoSQL database system and belongs to key-value type, whereas the Unity model does not. In order to add SQL feature for Redis database system, more efforts must be made.

Firstly, additional data indexing mechanisms need to be built for the Redis database system. Secondly, additional APIs need to be constructed based on these indexes.

| No | Features                                      | Unity          | MSI  |
|----|-----------------------------------------------|----------------|------|
| 1  | Language adopted for integration              | SQL            | SQL  |
| 2  | Parser of language                            | Based on JavaCC| Based on ANTLR |
| 3  | Accessing to multiple data sources simultaneously | Support       | Support |
| 4  | Access to MongoDB                             | Support        | Support |
| 5  | Access to MySQL                               | Support        | Support |
| 6  | Access to Redis                               | Not support    | Support |
| 7  | Optimization of arbitrary complex nested query | Not mentioned  | Support |
| 8  | Simplification of arbitrary complex conditional clauses | Not mentioned | Support |
| 9  | Support for the join operation of the database system itself | Support Partially | Support |
| 10 | Size of challenge                             | Relatively small| Relatively large |

Abbreviations: MSI, multiple sources integration; SQL, structured query language.
Finally, the parsed results of SQL must be mapped to these newly built APIs.

3. They are different in the optimization of arbitrarily complex nested queries. The MSI model supports this kind of optimization, whereas the Unity model does not mention the feature.

The MSI model uses Nest-G algorithm and BasicJoin algorithm to deal with this kind of optimization, in which the function of Nest-G algorithm is to convert arbitrary complex nested query into equivalent nonnested form that might include join predicate, and the join operation, which most likely involves multiple hybrid data sources, is processed using BasicJoin algorithm.

4. They are different in the simplification of arbitrary complex conditional clauses. The MSI model supports this kind of simplification, whereas the Unity model does not mention it.

In practical applications, arbitrary complex conditional clause, which usually appears in the form of a where clause, needs to be simplified and transformed. Espresso algorithm was adopted by the MSI model to finish this kind of work, which involves four main steps, including encode, expansion, simplification, and decode.

5. They are different in whether to support join operations of the specific database system itself. The Unity model supports the join operation of MySQL itself, whereas the MSI model also supports the left outer join operation of MongoDB database itself in addition to supporting the join operation of MySQL itself.

6. The challenge size that they face is different. From the aforementioned comparative descriptions, it is not difficult to find that the MSI model faces relatively greater challenges.

### 6.2 Quantitative analysis

According to the method described in this paper, an initial version of the system prototype is implemented using Java language. In this part, the performance of the initial prototype is quantitatively analyzed.

**1. Metrics and parameters**

In order to evaluate the performance of the model, several types of data operation are selected, and these operations in the SQL language are equivalent to the CRUD.

The basis of comparison is called single average response time, which was obtained by computing the average time of the response time consumed by 10 thousand data operations of the same type. At the same time, compared with the testing using the native APIs, the delay time of the data operations, represented by $\Delta T$, can be produced when the testing is performed using the APIs of the prototype system.

The software environment parameters are as follows:

- MongoDB DataStore: 3.2;
- Redis DataStore: 3.0;
- MySQL DataBase: 5.6;
- JDK: 1.7, 64 bit;
- OS: Red Hat Enterprise Linux 6.

Four computers are used for the deployment, respectively, of MySQL, MongoDB, Redis, and the prototype system, and one gigabit Ethernet switch is used for connecting these computers. The key parameters of each computer are as follows:

- CPU: Intel i5-6300(4 cores, 2.3 GHz);
- RAM: 8G(DDR3);
- Disk: 1T, 7200 rpm;
- Network card: gigabit Ethernet network card.

It should be noted that both MongoDB and Redis support horizontal extension technology, whereas MySQL itself does not have similar functionality. Since performance comparisons are required in data join evaluation, the evaluation involves only a single instance of each database system to compare fairly.

**2. Time analysis of accessing MongoDB**

One million and two million data records were loaded into the MongoDB database, respectively, in this evaluation. The evaluation results of access to MongoDB are shown in Figure 12. As can be seen from the figure, we have the following:

a. Compared with the native MongoDB API, the prototype system can produce some delay ($\Delta T$), but the value of the delay is not large. This is mainly caused by SQL’s parsing, optimization, and routing.
b. As more data records are loaded into MongoDB, the time to access MongoDB becomes longer. This is mainly due to more data records will lead to index maintenance time and index retrieval time increase.

3. Delay analysis of accessing MongoDB

The result of delay analysis for accessing MongoDB is shown in Figure 13, where Figure 13A shows the value of delay ($\Delta T$) and its variation, whereas Figure 13B shows the percentage of delay and its variation.

As can be seen from Figure 13, delays fluctuate in a range, that is, from 27us to 35us and from 3% to 4.2%. These delays is mainly caused by the parsing, optimization, and routing of SQL.

4. Time analysis of accessing Redis

One million and two million data records were loaded into the Redis database, respectively, in this evaluation. The evaluation results of access to Redis are shown in Figure 14. As can be seen from the figure, we have the following.
a. Compared with the native Redis API, the prototype system can produce some delay ($\Delta T$), but the value of the delay is not large. This is mainly caused by SQL’s parsing, optimization, and routing.

b. As more data records are loaded into Redis, the time to access Redis becomes longer. This is mainly due to more data records will lead to index maintenance time and index retrieval time increase.

5. Delay analysis of access to Redis

The result of delay analysis for accessing Redis is shown in Figure 15, where Figure 15A shows the value of delay ($\Delta T$) and its variation, while Figure 15B shows the percentage of delay and its variation.

As can be seen from Figure 15, delays fluctuate in a range, that is, from 25us to 32us and from 3.1% to 3.8%. These delays is mainly caused by the parsing, optimization, and routing of SQL.

6. Analysis of join operation

Taking left join as an example, the data join operation of the model is evaluated. The two database tables participating in the join evaluation are joined together by foreign key. Three scenarios were evaluated as follows.

a. The table in MySQL is joined to the table in MongoDB.

b. The table in MySQL is joined to the table in Redis.

c. The table in MongoDB is joined to the table in Redis.

Two million data records were loaded in MySQL, MongoDB, and Redis databases, respectively. In each scenario, in the two tables participating in the join test, the following amounts of data records actually participated in the join operation.

a. There are 1000 data records in each of the two tables participating in the join test, which are represented by “$\text{Rows} = 1K x 1K$”.

b. There are 2000 data records in each of the two tables participating in the join test, which are represented by “$\text{Rows} = 2K x 2K$”.

c. There are 3000 data records in each of the two tables participating in the join test, which are represented by “$\text{Rows} = 3K x 3K$”.

In addition, in order to avoid memory overflow, paging query is often used for data record query. Therefore, in the join evaluation, the page size parameter ($Sp$) mainly includes the following scenarios.

a. $Sp=1000$ is denoted by “$Sp=1K$”.

b. $Sp=2000$ is denoted by “$Sp=2K$”.

c. $Sp=3000$ is denoted by “$Sp=3K$”.

The evaluation of the join operation is shown in Figure 16. It can be seen from the figure the following.

a. Under the same conditions, that is, when the parameter $Sp$ and the number of data records participating in the join operation remain the same, it takes the minimum time that MongoDB is joined to Redis. This is mainly due to the fact that the two databases consume relatively little time in querying operations.

b. When parameter $Sp$ remains the same, the more data records participate in the join operation, the more time the join will consume. This is mainly due to the two factors. On the one hand, more records participate in join operation, which means that the number of loops in the join will increase. On the other hand, more records participate in join operation, which also results in more paging queries, this can also take some time.

c. When other conditions are invariant, increasing the value of parameter $Sp$ will reduce the time of those join operations in which the number of data records participating in join operation is bigger than parameter $Sp$. This is mainly due to the increase of parameter $Sp$, resulting in a reduction of the total number of paging queries.
It is important to note that, if some filtering conditions are applied to the data records in the tables, the number of data records actually participating in the join operation will be reduced, thus speeding up the join operation. In addition, if the data cache mechanism is adopted, the efficiency of join operation can be further improved. Therefore, in the follow-up study, there are still some optimization works to be completed.

6.3 Summary of evaluation

The following summarizations are made based on the aforementioned approach evaluation results.

1. Compared with similar integration approach, the approach proposed in this paper has comparative advantages in terms of the types of integrated data sources, the optimization of arbitrary complex nested queries, and the simplification of arbitrary complex conditional clauses.

2. Compared with the native APIs of NoSQL database system, including MongoDB and Redis, the operation delay ($\Delta T$) can be produced when using the SQL language to access these NoSQL database systems in the prototype system. However, the delay does not affect the user's practical feeling because it is relatively small in terms of the absolute value.

3. In order to access data from multiple hybrid data sources simultaneously, the basic join algorithm proposed in this paper also is feasible.

4. It is not only valuable but also feasible to add the SQL feature to these NoSQL database systems at the expense of some performance so as to improve the development efficiency of software systems.

7 CONCLUSION

The aim of this paper is to propose an effective integration approach of relational databases and NoSQL data-stores including MySQL, MongoDB, and Redis, which allows users to query data from both relational SQL systems and NoSQL systems in a single SQL query. The key contents were described including the model’s architecture and application patterns, the semantic transformation method between SQL and MongoDB API, the structures of the two kinds of auxiliary indexes of Redis data-store, and the querying method from multiple hybrid data-sources. In addition, some key algorithms were also given in the corresponding sections, and these algorithms are presented in the form of pseudo program code. Finally, the approach was evaluated, including the qualitative comparison and the quantitative analysis. The approach can effectively reduce development complexity and improve development efficiency of the software systems with multidatabases in cloud computing environment. This is the result of further research on the related topic, which fills the gap ignored by relevant scholars in this field to make a little contribution to the further development of the NoSQL technology.

Future works involve the optimization of algorithms and supporting for NoSQL’s transaction feature. In addition, authors also intend to support more NoSQL databases in the prototype system.

ACKNOWLEDGEMENTS

The authors would like to express that they are extremely grateful to the editors and the anonymous reviewers for their constructive comments, which have helped to improve the quality of this paper.
REFERENCES

1. Reyes-Delgado PY, Mora M, Duran-Limon HA, Rodriguez-Martinez LC, O’Connor RV, Mendoza-Gonzalez R. The strengths and weaknesses of software architecture design in the RUP, MSF, MBASE and RUP-SOA methodologies: a conceptual review. *Comput Stand Interface*. 2016;47(2):24-41.

2. Xie G, Zeng G, Chen Y, et al. Minimizing redundancy to satisfy reliability requirement for a parallel application on heterogeneous service-oriented systems. *IEEE Trans Serv Comput*. 2017;PP(99):1-16. https://doi.org/10.1109/TSC.2017.2665552

3. Xie G, Zeng G, Li R, Li K. Quantitative fault-tolerance for reliable workflows on heterogeneous IaaS clouds. *IEEE Trans Cloud Comput*. 2017;PP(99):1-14. https://doi.org/10.1109/TCC.2017.2780098

4. Chen W, Xie G, Li R, Bai Y, Fan Y, Li K. Efficient task scheduling for budget constrained parallel applications on heterogeneous cloud computing systems. *Futur Gener Comput Syst*. 2017;74(C):1-11.

5. Xie G, Zeng G, Jiang J, Fan C, Li R, Li K. Energy management for multiple real-time workflows on cyber-physical cloud systems. *Futur Gener Comput Syst*. 2017;PP(99):1-16. https://doi.org/10.1016/j.future.2017.05.033

6. Chen Y, Xie G, Li R. Reducing energy consumption with cost budget using available budget preassignment in heterogeneous cloud computing systems. *IEEE Access*. 2018;PP(6):20572-20583. https://doi.org/10.1109/ACCESS.2018.2825648

7. Pereira RHR, Perez-Schofield JBG, Ortin F. Modularizing application and database evolution - an aspect-oriented framework for orthogonal persistence. *J Softw Pract Exp*. 2017;47(2):193-221. https://doi.org/10.1002/spe.2415

8. Xie G, Chen Y, Liu Y, Fan C, Li R, Li K. JDAS: a software development framework for multidatabases. *J Softw Pract Exp*. 2018;48(2):366-382. https://doi.org/10.1002/spe.2472

9. Carlson JL. *Redis in Action*. Greenwich, CT: Manning Publications Co; 2013.

10. Chodorow K. *MongoDB: The Definitive Guide*. Sebastopol, CA: O’Reilly Media Inc; 2013.

11. George L. *HBase: The Definitive Guide*. Sebastopol, CA: O’Reilly Media Inc; 2011.

12. Vukotic A, Watt N, Abedrabbo T, Fox D, Partner J. *Neo4j in Action*. Greenwich, CT: Manning Publications Co; 2014.

13. Lawrence R. Integration and virtualization of relational SQL and NoSQL systems including MySQL and MongoDB. In: Proceedings of the 2014 International Conference on Computational Science and Computational Intelligence (CSCI); 2014; Las Vegas, NV.

14. Pokorny J. NoSQL databases: a step to database scalability in web environment. *Int J Web Inf Syst*. 2013;9(1):69-82.

15. Ickert F, Del Fabro MD, de Almeida EC, Scherzinger S. NoSQL data model evaluation on app engine datastore. In: Proceedings of the 28th Brazilian Symposium on Databases; 2013; Recife, Brazil.

16. Cruz F, Gomes P, Oliveira R, Pereira J. Assessing NoSQL databases for telecom applications. In: Proceedings of the 2011 IEEE 13th Conference on Commerce and Enterprise Computing (CEC); 2011; Luxembourg, Luxembourg.

17. Shao B, Conrad TO. Are NoSQL data stores useful for bioinformatics researchers. *Int J Recent Innov Trends Comput Commun*. 2015;3(3):1704-1708.

18. Xiong W, Hawley D, Monismith D. NoSQL in database education: incorporating non-relational concepts into a relational database course: panel discussion. *J Comput Sci Coll*. 2015;30(5):151-152.

19. Costa C, Santos MY. Reinventing the energy bill in smart cities with NoSQL technologies. In: *Transactions on Engineering Technologies*. Singapore: Springer Science+Business Media; 2016:383-396.

20. Han J, Hahng E, Le G, Du J. Survey on NoSQL database. Paper presented at: 2011 6th International Conference on Pervasive Computing and Applications; 2011; Port Elizabeth, South Africa.

21. Bhuvan NT, Elayidom S. A technical insight on the new generation databases: NoSQL. *Int J Comput Appl*. 2015;121(7):24-26.

22. Cattell R. Scalable SQL and NoSQL data stores. *ACM SIGMOD Rec*. 2010;39(4):12-27.

23. Liao Y-T, Zhou J, Lu C-H, et al. Data adapter for querying and transformation between SQL and NoSQL database. *Futur Gener Comput Syst*. 2016;65(12):111-121.

24. Vilaça R, Cruz F, Pereira J, Oliveira R. An effective scalable SQL engine for NoSQL databases. In: *Distributed Applications and Interoperable Systems: 13th IFIP WG 6.1 International Conference, DAIS 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DiCoTec 2013, Italy, June 3-5, 2013. Proceedings*. Berlin, Germany: Springer; 2013:155-168.

25. Tattemura J, Po O, Hsiung W-P, Hacigümüş H. Partiqle: an elastic SQL engine over key-value stores. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD)*; 2012; Scottsdale, AZ.

26. Khan S, Mane PV. SQL support over MongoDB using metadata. *Int J Sci Res Publ*. 2013;3(10):1-5.

27. Rith J, Lehmayer PS, Meyer-Wegener K. Speaking in tongues: SQL access to NoSQL systems. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC); 2014; Gyeongju, South Korea.

28. Calli A, Mello RDS. SimpleSQL: a relational layer for simple DB. In: *Advances in Databases and Information Systems: 16th East European Conference, ADBIS 2012, Poznañ, Poland, September 18-21, 2012. Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2012:99-110.

29. Liu ZH, Hammerschmidt B, McMahon D, Liu Y, Chang HJ. Closing the functional and performance gap between SQL and NoSQL. In: Proceedings of the 2016 International Conference on Management of Data (SIGMOD); 2016; San Francisco, CA.
30. Tahara D, Diamond T, Abadi DJ. Sinew: a SQL system for multi-structured data. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD); 2014; Snowbird, UT.

31. Zhang H, Wang Y, Han J. Middleware design for integrating relational database and NoSQL based on data dictionary. In: Proceedings 2011 International Conference on Transportation, Mechanical, and Electrical Engineering (TMEE); 2011; Changchun, China.

32. Hacigümüş H, Tatemura J, Hsiung W-P, et al. CloudDB: one size fits all revived. Paper presented at: 2010 6th World Congress on Services; 2010; Miami, FL.

33. Atzeni P, Bugiotti F, Rossi L. Uniform access to non-relational database systems: the SOS platform. In: Advanced Information Systems Engineering: 24th International Conference, CAiSE 2012, Gdansk, Poland, June 25-29, 2012. Proceedings. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2012:160-174.

34. Sellami R, Bhiri S, Defude B. ODBAPI: a unified REST API for relational and NoSQL data stores. Paper presented at: 2014 IEEE International Congress on Big Data; 2014; Anchorage, AK.

35. Kim W. On optimizing an SQL-like nested query. ACM Trans Database Syst. 1982;7(3):443-469.

36. Ganski RA, Wong HKT. Optimization of nested SQL queries revisited. In: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD); 1987; San Francisco, CA.

37. Parr T. The Definitive ANTLR 4 Reference. Raleigh, NC: The Pragmatic Programmers; 2012.

38. Brayton RK, Hachtel GD, McMullen CT, Sangiovanni-Vincentelli AL. Logic Minimization Algorithms for VLSI Synthesis. Hingham, MA: Kluwer Academic Publishers; 1984.

39. MongoDB Inc. MongoDB documentation (release 3.2.4). New York, NY: MongoDB Inc; 2016.

40. Bayer R, McCreight E. Organization and maintenance of large ordered indexes. Acta Informatica. 1972;1(3):173-189.

How to cite this article: Li C, Gu J. An integration approach of hybrid databases based on SQL in cloud computing environment. Softw: Pract Exper. 2019;49:401–422. https://doi.org/10.1002/spe.2666