Autom Softw Eng (2014) 21:489–508
DOI 10.1007/s10515-013-0135-x

JackHare: a framework for SQL to NoSQL translation using MapReduce

Wu-Chun Chung · Hung-Pin Lin · Shih-Chang Chen · Mon-Fong Jiang · Yeh-Ching Chung

Received: 15 December 2012 / Accepted: 6 September 2013 / Published online: 28 September 2013

Abstract As data exploration has increased rapidly in recent years, the datastore and data processing are getting more and more attention in extracting important information. To find a scalable solution to process the large-scale data is a critical issue in either the relational database system or the emerging NoSQL database. With the inherent scalability and fault tolerance of Hadoop, MapReduce is attractive to process the massive data in parallel. Most of previous researches focus on developing the SQL or SQL-like queries translator with the Hadoop distributed file system. However, it could be difficult to update data frequently in such file system. Therefore, we need a flexible datastore as HBase not only to place the data over a scale-out storage system, but also to manipulate the changeable data in a transparent way. However, the HBase interface is not friendly enough for most users. A GUI composed of SQL client application and database connection to HBase will ease the learning curve. In this paper, we propose the JackHare framework with SQL query compiler, JDBC driver and a systematical method using MapReduce framework for processing the unstructured data in HBase. After importing the JDBC driver to a SQL client GUI, we can exploit the HBase as the underlying datastore to execute the ANSI-SQL queries. Experimental results show that our approaches can perform well with efficiency and scalability.

W.-C. Chung · H.-P. Lin · S.-C. Chen (✉) · Y.-C. Chung
Dept. of Computer Science, National Tsing Hua University, Hsinchu 300, Taiwan
e-mail: shcchen@sslab.cs.nthu.edu.tw

W.-C. Chung
e-mail: wcchung@sslab.cs.nthu.edu.tw

H.-P. Lin
e-mail: tlcsmall3@sslab.cs.nthu.edu.tw

Y.-C. Chung
e-mail: ychung@cs.nthu.edu.tw

M.-F. Jiang
is-land Systems Inc., Hsinchu Science Park, 3F, No.4, Prosperity Rd. 2, Hsinchu 300, Taiwan
e-mail: mike@is-land.com.tw

Keywords Cloud computing · Unstructured data processing · MapReduce · NoSQL database · HBase · JDBC · Compiler

1 Introduction

With the size of datasets growing rapidly, the large-scale data processing has become more and more important. In early years, there were many works that focused on finding solutions to solve the storage and processing the massive data. Using parallel relational database management systems (parallel DBMSs) is just one of the solutions. Since the techniques (Chang et al. 2008; Dean and Ghemawat 2008; Ghemawat et al. 2003) Google had published, the large-scale data processing steps into a new pattern.

Generally, accessing relational database (RDB) is one of the most popular ways to manage data for enterprises. RDB techniques are mature and the performance is good for small datasets. With rapidly growing of data being collected and analyzed, RDB is no more a good solution. Users store large-scale of data in different tables in RDB, it becomes complicated and costs more time to access data. The ways cloud computing provides users to access datastore usually facilitate the operations of separating tables or distributing files to different nodes. In this manner, a submitted query can be processed in parallel to reduce the turnaround time. However it is inconvenient for users to access the datastore in cloud once the way they access data is changed. Moreover, changing datastore from RDB to cloud datastore requires steep learning curve to users.

To avoid the steep learning curve for users, one solution is to develop a framework which provides a front-end interface as shown in Fig. 3 to submit ANSI-SQL queries and a back-end to employs NoSQL database for accessing large-scale data. Thus the ANSI-SQL users can access NoSQL database without having to learn new programming languages.

MapReduce is one of these techniques acted as a large-scale data processing platform, which is initially used to analyze the text data and build the text index. The Clydesdale work (Kaldewey et al. 2012) points out several attractive properties compared to parallel DBMSs, and also states the importance of structured data processing. As examples, scalability, fault-tolerance and flexible programming framework are important features required on large-scale data processing. Although the parallel DBMS is one of the solutions to process the structured data, how to process the large-scale structured data in a scalable way is a big challenge.

A comparison of approaches to large-scale data analysis (Pavlo et al. 2009) evaluated the performance comparisons between parallel DBMSs and MapReduce. That research pointed out several problems for data processing using MapReduce. However, making DBMSs more scalable is not easy according to the rules (Stonebraker and Cattell 2011). Developers have to implement functions in the application-level and implement the mechanism of replication or sharding. In this way, the solutions may highly depend on the application itself. FunSQL (Binnig et al. 2012) is functional language to extend SQL while statements can be optimized and parallelized. Therefore, more complex application logic can be developed as in SQL. Oracle has supported for MapReduce paradigm through specific mechanism. Native Hadoop programs cannot be executed in this framework without being rewritten. A prototype (Su and Swart 2012) for such framework was proposed to seamlessly integrate MapReduce and SQL. This explains that MapReduce is a better solution for a wide variety of applications while Big Data is involved.

In recent years, several studies work on adopting the MapReduce to perform the unstructured data processing, e.g. HadoopDB (Abouzeid et al. 2009), Pig Latin (Olston et al. 2008) and Hive (Thusoo et al. 2009). However, most of previous works pay less attention to address the adaption of frequently updatable datastore such as Bigtable or HBase (Apache HBase (2013). http://hbase.apache.org). In some enterprises' needs, the data resided in a database may be frequently changed as the update occurs. Accordingly, we need a flexible datastore not only to place the data over a scale-out storage system, but also to manipulate the changeable data in a transparent way. Based on the flexible datastore, we can further process the large-scale unstructured data on MapReduce in a scalable manner.

In this paper, we propose the JackHare framework to facilitate the process of accessing large-scale datasets. The JackHare framework provides a front-end interface and back-end datastore connection allowing users to use the ANSI-SQL queries to manipulate large-scale data. The front-end of the JackHare framework includes a JDBC driver which implements the Java database connectivity and provides a compiler to scan, parse and execute ANSI-SQL queries. In order to generate code, we provide a systematical method using MapReduce for the unstructured data processing in NoSQL database. To support an updatable datastore, we exploit the HBase as the underlying NoSQL database. Therefore, a JDBC driver for the connection to HBase is provided in this framework for users to access data in HBase.

Based on the HBase, we analyze the major manipulation languages of the ANSI-SQL and provide the corresponding implementations to manipulate the data residing in the NoSQL database. To manipulate the datasets effectively, a data placement model which is aligned with the target NoSQL database is a critical issue. Therefore, to organize the data with less complexity, we also introduce a mapping strategy to translate the data model from the relational database to the NoSQL database. With the corresponding manipulations, developers can exploit transparent programming interfaces to process the unstructured data in NoSQL database. Moreover, we further conduct the unstructured data processing on MapReduce to scale the data manipulation among the large-scale data.

To use the JackHare framework is to import the JDBC driver to a SQL client, e.g. SQuirreL. After setting up the configuration in SQL client, submitting ANSI-SQL queries through the SQL client can manipulate the large-scale data in HBase. With our work, we can approach the scalable data processing in NoSQL database without any change to the legacy SQL manipulations. Experimental results show that our approaches perform the effectiveness of the efficiency and the scalability with the variety of SQL queries.

The rest of this paper is organized as follows. Section 2 reviews the related works, while Sect. 3 introduces the JackHare framework. Section 4 presents the data model and SQL implementations using MapReduce. Section 5 shows the experimental results to illustrate the system performance while Sect. 6 concludes this paper.

2 Related work

Some works have been done for massive data processing in recent years. The most similar works we focus on are the structured data processing on MapReduce including Pig (Olston et al. 2008), Hive (Thusoo et al. 2009), YSmart (Lee et al. 2011), S2MART (Gowraj et al. 2013), HadoopDB (Abouzeid et al. 2009), and Clydesdale (Kaldewey et al. 2012).

Some of previous works provide high level query languages, e.g., SQL-like queries, for data processing without directly using the MapReduce programming. Their contribution is to facilitate the use of MapReduce framework for users. Instead of learning MapReduce framework, users spend less time learning SQL-Like languages. As well-known examples, Pig introduces a simpler procedural language named Pig Latin while Hive presents a declarative language called HiveQL. These two languages are all built on top of the MapReduce framework. Compared to writing MapReduce code, coding with the Pig Latin is relatively simple. However, the logic of data manipulation may differ from the SQL query, whereas HiveQL is more comfortable with SQL. Moreover, Hive presents a data warehouse built on top of Hadoop distributed file system (HDFS) (Apache Hadoop (2013). http://hadoop.apache.org) for storing the data originally resided in a relational database. Nevertheless, executing complex SQL queries with HiveQL is still difficult. QMapper (Xu and Hu 2013) was then proposed to address this problem by utilizing the rewriting rules and MapReduce flow evaluation, and improved the performance of Hive significantly. YSmart pointed out that Hive should consider the correlations of queries for better performance. Once the correlations are considered, redundant computations, I/O operations and network transferring can be significantly reduced. S2MART tried to transform SQL queries into MapReduce jobs and reduce both data transfer cost and network transfer cost. The authors also provided comprehensive study about various features of this research. In our work, we focus on data processing with the logic of SQL queries in NoSQL database, e.g., HBase. We present a SQL implementation on MapReduce in HBase for processing and storing the data.

The architecture of HadoopDB is a cluster composed of the MapReduce framework among multiple single-node relational databases. The data in HadoopDB is stored in the relational databases instead of the HDFS. The contribution of this work is to approach the performance of parallel DBMSs and sustains the scalability and fault tolerance as Hadoop. In addition, HadoopDB provides SQL query via a translation called SQL-MR-SQL (SMS), which is an extended translation layer based on Hive. The translation flow of Hive is to translate the HiveQL into MapReduce for processing the files located in the HDFS. HadoopDB modifies and extends this translation flow to change the interaction target to each single-node relational database. However, it is a complex architecture to overcome the scalability problem of traditional database systems. In our work, we provide an alternative solution without modifying the architecture of Hadoop.

Instead of developing the high level query languages, Clydesdale (Kaldewey et al. 2012) presents a prototype system for structured data processing on MapReduce. The proposed system provides a significant performance improvement compared to existing works and does not need any modification of Hadoop implementations. Clydesdale focuses on processing the data fitting a star schema. On the contrary, our work addresses the data manipulation of SQL queries. Besides, we implement the SQL manipulations on top of a NoSQL database instead of the HDFS.

To sum up, the previous works mainly focus on using MapReduce for the data analysis and the performance improvement over HDFS. However, these works are not desirable for processing the data which may be changed frequently, because of the write-once-read-many feature of HDFS. In our work, we intend to provide a solution that can fulfill the frequent update of the data. Consequently, HBase is employed as our datastore to support changeable data records while sustaining the scalability and reliability in NoSQL database. Moreover, to organize the datasets, a simpler data model is designed to transform the data schema between the relational database and the HBase. We further conduct MapReduce algorithms on SQL implementations to efficient manipulate the datasets in a scalable NoSQL database.

3 The JackHare framework architecture

To facilitate the use of MapReduce and HBase for ANSI-SQL users, the JackHare framework composed of front-end and back-end to translate ANSI-SQL queries and perform logical operations while accessing HBase. JackHare aims to ease the learning curve of analyzing large-scale datasets in NoSQL database instead of RDB. Figure 1 shows the architecture of JackHare framework. The front-end includes a GUI, compiler and JDBC driver while the back-end includes HBase which is a scalable NoSQL database. Users can choose their favorite application as a GUI, e.g. SQuirreL SQL client, to import the JDBC driver. The JDBC driver includes an ANSI-SQL query

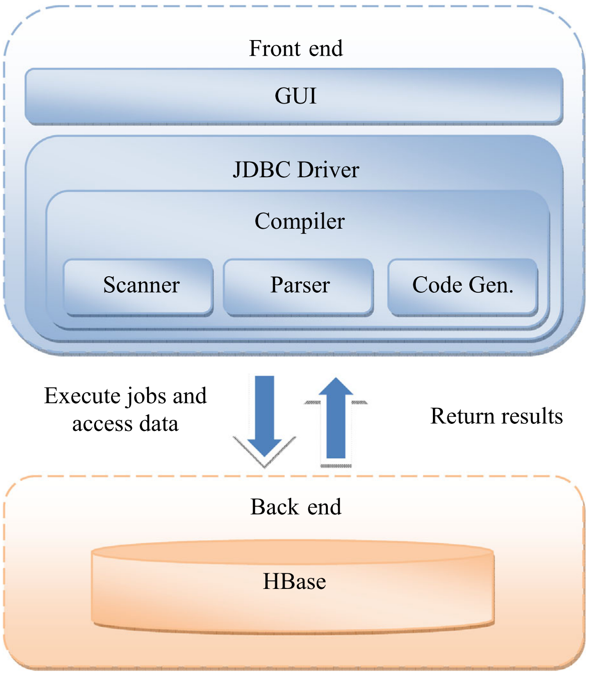


Fig. 1 The architecture of JackHare framework

compiler to scan, parse and execute queries. The scanner, parser and code generator are implemented using the open source project, Druid. While an ANSI-SQL query submitted, the scanner breaks queries into tokens. The parser manipulates the abstract syntax tree which is comprised of tokens. The code generator produces MapReduce jobs to access HBase and then perform logical operations according to the parsed commands. The results of ANSI-SQL queries are returned and shown on GUI.

The data models of tables in HBase are quite different from those in RDB. In this paper, we define the data model used for HBase and introduce it in Sect. 4.1. The relation between tables in RDB and HBase should be given for the code generator to precisely retrieve the information of the right column in right table to produce MapReduce job. The JackHare framework chooses Apache Derby as the metadata server. Derby stores the table name, column families and column qualifiers of HBase which can be mapped to database, table name and column name of RDB.

The process of operations from submitting an ANSI-SQL query to returning the results on SQL client interface is as follows.

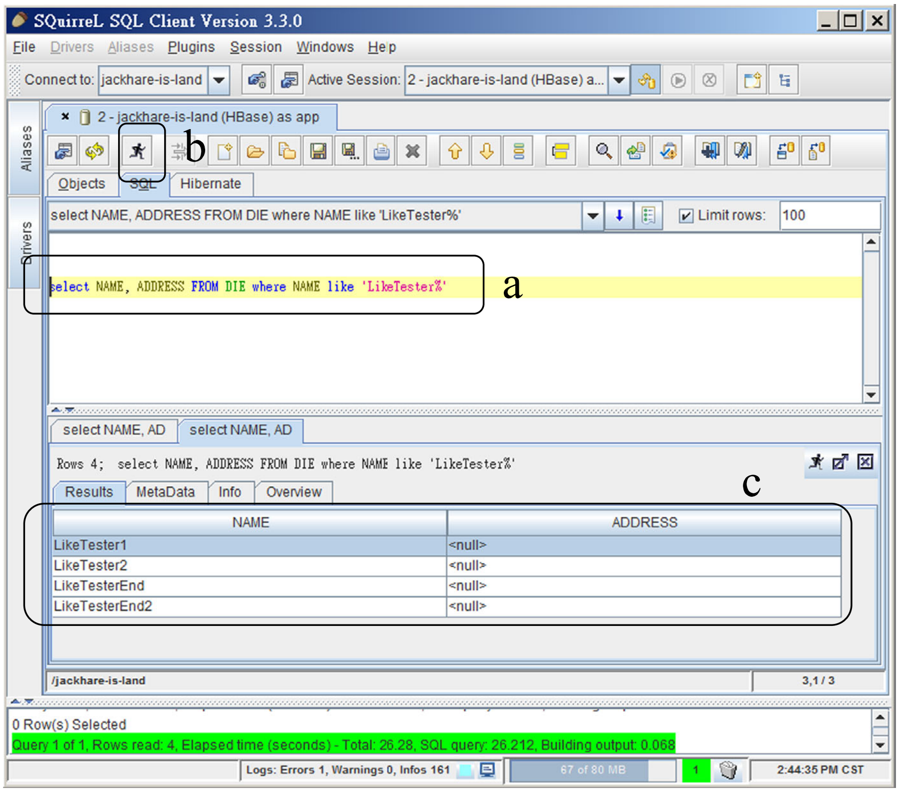
- User submits an ANSI-SQL query by SQL client application.
- The compiler scans and parses the ANSI-SQL query.
- Lookup the related table name, column families and column qualifier of HBase.
- Generate MapReduce code according to the query commands and metadata.
- Access HBase and execute the MapReduce job.
- The results wrapped back from the back-end.
- The returned results are shown on SQL client application according to RDB schema.

In this paper, SQuirreL is used as the user interface in frond-end. It allows users to access NoSQL service with familiar RDBMS interface. Figure 2 is the screen capture of SQuirreL with three blocks indicating important information. Block ‘a’ shows the space to place user’s ANSI-SQL query while block ‘b’ is a button for executing the query by SQuirreL. Lastly, block ‘c’ shows the results regarding to the RDB schema after finishing a series of operations including scanning, parsing, code generating, MapReduce job executing.

To produce MapReduce jobs for ANSI-SQL queries, we need the data model to store data from RDB, the mechanism of analysis ANSI-SQL clauses and the design of MapReduce algorithm. Next section describes a high level view for data model, systematic flow for clauses translating, and the design of MapReduce execution plan and algorithms.

4 Unstructured data processing in HBase

This section describes the methods we propose for structured data processing in HBase. At first, we describe how to remap the data in relational database to HBase. Then, we analyze ANSI-SQL to build up methods with a systematic flow. Finally, we propose a series of data manipulation methods with MapReduce framework to process the massive data in HBase.


SQuirreL SQL Client Version 3.3.0
File Drivers Aliases Plugins Session Windows Help
Connect to: jackhare-is-land
Active Session: 2 - jackhare-is-land (HBase) a...
2 - jackhare-is-land (HBase) as app
Aliases
Drivers
Objects SQL Hibernate
Limit rows: 100
b
a
c
select NAME, AD
select NAME, AD
Results MetaData Info Overview
/jackhare-is-land
3,1 / 3
Logs: Errors 1, Warnings 0, Infos 161
67 of 80 MB
2:44:35 PM CST

```
select NAME, ADDRESS FROM DIE where NAME like 'LikeTester%'
```

Rows 4; select NAME, ADDRESS FROM DIE where NAME like 'LikeTester%'

NAME	ADDRESS
LikeTester1	<null>
LikeTester2	<null>
LikeTesterEnd	<null>
LikeTesterEnd2	<null>

```
0 Row(s) Selected
Query 1 of 1, Rows read: 4, Elapsed time (seconds) - Total: 26.28, SQL query: 26.212, Building output: 0.068
```

Fig. 2 The screen capture of SQuirreL

4.1 Data model

Data model is a high level view of data organization and manipulation, which deeply affects the methods of data store and access. To perform the remap process, we have to understand the differences of data model between relational database and HBase. After that, a corresponding approach is presented to transform the data from relational database to HBase and is fulfilled to our proposed JOIN algorithm in Sect. 4.2.3.

According to different kinds of data models, NoSQL databases can be classified into different categories, and HBase is a Bigtable-style database (Chang et al. 2008). Table 1 compares some critical aspects between the Bigtable-style and the relational data model. The column-/row-oriented represents the physical placement of datasets. In this work, we do not concern this criterion because the physical placement may affect the performance of the queries case by case. Instead, we focus on developing a generic and systematic solution. Regarding the dimension and the cell version, the Bigtable data model can support multiple dimensions and cell versions, whereas the supportability of a relational data model is limited. In fact, the cell version is a hidden dimension to record the histogram of a cell, which is not considered in the design of a relational data model.

Table 1 Difference between Bigtable data model and relational data model

	Bigtable data model	Relational data model
Column-/Row-oriented	Column	Row
Dimension	Multi-dimension	Two-dimension
Cell version	Multiple version	One version
Number of columns	Arbitrary	Fixed
Group of columns	Yes (Column Family)	No

Col. Family / Row Key	DIE			LOT			WAFER			RowKeys		
	DATE	…	LOT	HBIN	…	SBIN	LOT_ID	…	SLOT_NO	DIE	LOT	WAFER
DIE:A0000.00	#$#	…	#$#									
…	…	…	…									
DIE:A9999.99	@#	…	%#$									
LOT:A0000.00				@#@	…	$#$						
…				…	…	…						
LOT:A9999.99				@^^	…	#!%						
WAF:A0000.00							@#@	…	#$@			
…							…	…	…			
WAF:A9999.99							#^$@	…	@^*			
RowKeyList										@#!	%&^	$%#

Fig. 3 Data model of a big table transformed from relational tables of a relational database

In the Bigtable data model, the number of columns is arbitrary, whereas the number of columns is fixed in the relational data model. While the fixed number of columns is well designed to deal with the structured data, pre-deciding the schema of table is difficult to handle the unstructured data. Hence, the Bigtable-style data model utilizes an arbitrary number of columns to store such datasets. Moreover, the dimension and the group concept are two properties that mostly affect the remapping design. In the matter of dimension, we remap the two dimensions of relational database to the rows and the qualifiers of HBase. Each row key of HBase is a combination of table name and primary key in the original relational table. Besides, the column family, which is the group concept of HBase, is applied to identify each relational table of a relational database. Additionally, we adopt one special column family and a row for the JOIN algorithm. With this mapping strategy, those tables of an application in a relational database can be successfully transformed into a "big table" of HBase.

As illustrated in Fig. 3, we use the dataset from a semiconductor industry to describe above remap process. The table of HBase consists of four column families, in which DIE, WAFER, and LOT represent three original relational tables respectively. Therefore, when there is a query related to table DIE in the view of relational database, we access the data related to column family DIE in the table in HBase for this query. A special column family "RowKeys" is defined as storing the row key

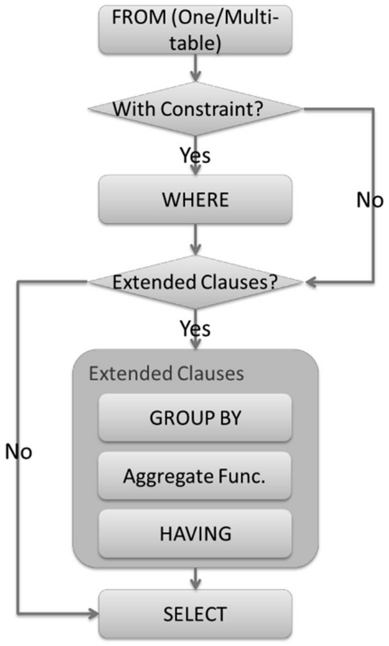


Fig. 4 Logical flow of a basic SQL statement

list generated by our JOIN algorithm. Section 4.2.3 describes more about the use of "RowKeys" in (c) Joining query in (3) Extended clauses.

4.2 Analysis of SQL clauses

The SELECT statement is the basic scope of query support in this paper. It means the SQL language begins with the SELECT keyword and is composed of clauses. To design methods with a systematic flow, we need to translate each clause of the SELECT statement into an execution plan of one or multiple MapReduce phases. Subsequently, we describe two steps for our proposed translation and summarize the design of these clauses into a logical flow as shown in Fig. 4.

4.2.1 SELECT, FROM and WHERE clauses

This work focuses on the translation of three basic clauses of SELECT statement: SELECT, FROM, and WHERE. Firstly, we describe how to translate these clauses into a logical flow, and then, we translate this logical flow into the MapReduce execution plan.

Queries are usually given with an order: SELECT, FROM, and then WHERE. But the data manipulation flow is definitely not in this order. We have to reorder the process of a basic SELECT statement. The first step is to deal with the input of a query. No matter how many input tables the FROM clause has to access, we filter all the input tables if the query statement contains a WHERE clause. The WHERE clause specifies the condition which rows of the input tables can be used. In other words, the WHERE clause is not necessary for every query. If the WHERE clause is given, all the rows of input tables are used. The last step is to output the data fields which are specified in the SELECT clause. Hence, as show in Fig. 4, the data manipulation order of these three clauses is FROM, WHERE (optional), and then SELECT.

4.2.2 Extended clauses

Apart from three basic clauses as mentioned above, the SELECT query has some extended capabilities. These extended clauses are used to perform some advance processing based on the result of basic clauses. We introduce these capabilities according to the clauses and features as follows.

(a) *GROUP BY*:
GROUP BY is a clause that specifies a column for the grouping. The AGGREGATE FUNCTIONs is usually given with a GROUP BY clause to compute the result of rows in a group. Then, a constraint of the HAVING clause can be further set to filter the query result. Basically, without other clauses, the query result of each group is only a single row.

(b) *HAVING*:
HAVING is a clause that specifies a constraint to filter the result, which is often cooperated with GROUP BY and AGGREGATE FUNCTIONs. If the GROUP BY clause is given, the HAVING clause is resolved after handling the GROUP BY clause. In the case of AGGREGATE FUNCTIONs, the HAVING clause is executed after the WHERE clause or the FROM clause if the WHERE clause is not given.

(c) *ORDER BY*:
ORDER BY is a clause that specifies a column for the sorting. The ORDER BY clause has to be the last clause of a SELECT statement. The query result would be sorted if ORDER BY clause is given.

(d) *JOIN*:
JOIN is a clause that specifies the number of input tables is more than one. When the FROM clause of a query is given with more than one table, a JOIN operation could be identified. A JOIN query can cooperate with key words, e.g., ON, INNER, to perform different kinds of joining queries. In current work, we focus on the CROSS JOIN and the EQUAL JOIN (i.e., JOIN … ON.) to construct a systematic flow of proposed methods.

(e) *AGGREGATE FUNCTIONs*:
An AGGREGATE FUNCTION is the operation to summarize the value of rows of a given column in a group or a full table. The aggregate functions often cooperate with the GROUP BY clause. The ANSI-SQL standard defines many kinds of aggregate functions. In current work, we focus on five commonly used functions. SUM and AVG functions will work with the numeric. While SUM sums up the value of a specified column, AVG computes the average value of a specified column. On the other hand, the COUNT function is used to calculate the number of rows in a given column. MIN and MAX are used to find out the minimum and the maximum value of a specified column.

Next, we take the following query as an example:

SELECT WAFER_ID, **SUM**(PSVDDV)
FROM DIE
WHERE WAFER_ID **LIKE** 'A00002.2 %'
GROUP BY WAFER_ID
HAVING SUM(PSVDDV) > 2000000

In this example, the input table DIE which specified by the FROM clause is filtered firstly according to the constraint of "WAFER_ID LIKE 'A00002.2 %'". Secondly, the GROUP BY clause groups and summarizes the rows based on common values in a set of WAFER_ID. One group is treated as a single row by setting an SUM function. The SUM function is a kind of aggregate function to compute the arithmetic sum in a set of grouped rows. Then, the HAVING clause filters the grouped rows that satisfied with "SUM(PSVDDV) > 2000000". Finally, the SELECT clause specifies WAFER_ID and SUM(PSVDDV) to be retrieved as the output format.

According to the sample query, we can successfully translate the specified clauses into a logical flow as shown in Fig. 4. There are other two clauses we also take into consideration, which are ORDER BY and JOIN. Since the ORDER BY clause is performed to sort the result of a specific column, we develop this clause after other clauses. On the other hand, when the FROM clause appears in more than one table, a joining query occurs. Subsequently, we describe translating the logical flow into MapReduce execution plan.

4.2.3 Design of MapReduce algorithm

We have described how to translate the SQL clauses into a logical flow. This section focuses on the design of MapReduce execution plan and corresponding algorithms.

(1) *From logical flow to MapReduce execution plan*

To explain the translation from the logical flow to a MapReduce execution plan, we give a brief review of Hadoop MapReduce at first. A Hadoop MapReduce job is typically divided into two major phases. In the Map phase, the Mapper processes the input data and mapping the data into corresponding key-value sets. Then, the framework steps into the shuffle/sort stage. In this stage, the output of key-value sets from the Map phase is redistributed by keys to achieve a global data organization. Finally, the last Reduce phase combines the received key-value sets into an ordered list according to the keys. In this phase, Reducer processes the key-value sets with the same key to perform a global combination.

Since we get the logical flow of the SQL statement, we can take the translation process as a mapping between the logical flow and the MapReduce execution stages. Figure 5 summarizes the relation between them. For most of the basic statements, one Mapper is involved to resolve the query; at the meanwhile, the Reducer is not involved. In the case of giving the extended clauses of SQL statement, the remaining logic flow of the query is executed by the Reducer, e.g. GROUP BY, AGGREGATE FUNCTIONs and HAVING. However the ORDER BY clause is not executed by the Reducer since the row keys in HBase are sorted by the type of byte array. The ORDER BY clause can exploit this feature to retrieve the sorted result. Once the JOIN clause is involved, we run a MapReduce job to create the joined table at first. The remaining

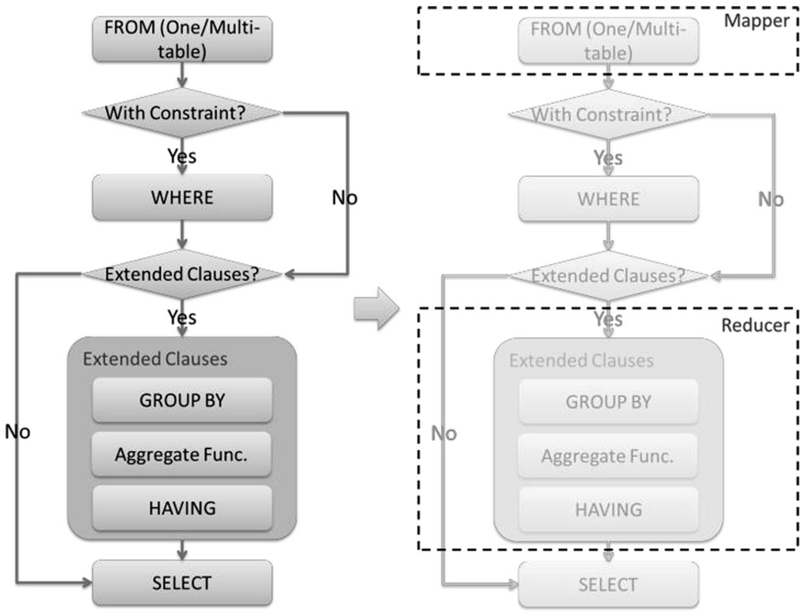


Fig. 5 Logic flow to MapReduce plan

clauses are performed by the other MapReduce job to resolve the query. Thus, the number of MapReduce jobs could be more than one depending on the complexity of the query.

After introducing the MapReduce execution plan, we further describe how to conduct the combinations of SQL clauses on MapReduce in HBase.

(2) *Basic clauses of a SQL statement*

Given a basic clause, the WHERE clause can simply be treated as a filter. In this work, regarding the MapReduce framework in HBase, the Filter instance of the HBase system is used to filter the data in HBase. To exploit the HBase filter, we use the Scan instance to filter more results of matched rows. In the scanning process, we add a Filter on the Scan instance to filter the input table. The Filter instance can be customized according to various constraints. After scanning with the Filter, the result is sent to the Mapper. In this way, the number of data sent to the Mapper is smaller than filtering the data in the Map phase. Moreover, the scanning process could be parallelized and so does the corresponding Filter operation.

After receiving the result from the Scan instance, Mapper uses the TableInputFormat to convert the tabular data into a format that can be identified by the MapReduce framework. The TableInputFormat is a kind of input format provided by Hadoop MapReduce. Using the TableInputFormat, the input of Map function in Mapper is filtered and read one row at a time. The Map function only needs to get the columns which SELECT clause specified and output to the result table in HBase. In this manner, we can retrieve the data input and output in parallel. Thus, when the query is simply using the basic SELECT statement, we need only the Map phase to get query result. There is no operation for reduce phase in this situation.

(3) *Extended clauses*

For easy understanding, we classify the combinations of extended clauses into four categories and explain the corresponding designs of MapReduce algorithms.

(a) *Grouping query*: The grouping query is a combination of the clause with GROUP BY, AGGREGATE FUNCTIONs or HAVING. The MapReduce framework essentially provides the ability to process the GROUP BY clause. Accordingly, in our work, the Mappers assign specified keys for grouping and the Reducers group the key-value pairs with the same key into a group. Since the Reducers primitively receive the same key from the output of Mappers, all we have to do is to decide the key-value pairs for each row.

In the Map phase, we get some related information from the MapReduce configuration instance at first. Secondly, the following steps are processed in the Mapper. Whether the GROUP BY clause of a query cooperates with or without HAVING or AGGREGATE FUNCTIONs, the key of the key-value pair is the specified value of the column which is given in the GROUP BY clause. To support AGGREGATE FUNCTIONs, the value of the key-value pair is the combination of the column name and the values specified by the query. After constructing the key-value pair, the Mapper outputs this pair to the corresponding Reducer which is responsible for this key.

Once Reducers collect all key-value pairs with the same key, they start to summarize the values according to the specified AGGREGATE FUNCTIONs. Each column may be required to perform multiple AGGREGATE FUNCTIONs. We can get this information from the MapReduce configuration. According to the part of column name, Reducer can distinguish which column the value is belonged to. Then, Reducer knows what aggregate operations to be performed to this value. After every column of the row is processed, the Reducer writes the result to a result table in HBase. Besides, if the HAVING clause is given, the Reducer will check if the result satisfied the condition specified by the HAVING clause.

(b) *Aggregate query*: The aggregate query is a combination of the clause with AGGREGATE FUNCTIONs. We implement these functions in the Reduce phase. When the aggregate function does not cooperate with GROUP BY clause, the result of the query is summarized to a single row. Therefore, the key of the key-value pair is set to the same value and sent to the same Reducer.

(c) *Joining query*: The joining query is a combination of the clause with JOIN. The ANSI-SQL standard defines a variety of joining query. In this paper, we consider both the equal join and the cross join of two tables. Several existing works have introduced their join algorithm using MapReduce (Afrati and Ullman 2010; Blanas et al. 2010; Okcan and Riedewald 2011). The comparison of join algorithm (Blanas et al. 2010) analyzes different join algorithms and presents a comprehensive comparison of those algorithms. However, the previous algorithms focus on the equal join. Our proposed method can extend an extra job to filter the result of cross join to achieve the equal join.

To support the cross join, we construct a row key list of the smaller input table by a MapReduce job and store the list in the HBase at first. For the process of joining, we create another MapReduce job which only contains a Map phase. Figure 6 illustrates flow in the Map phase in detail. To produce more parallel processing operations, we

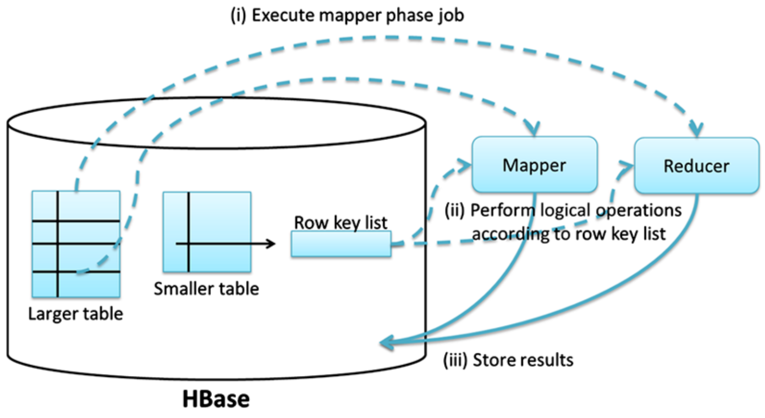

(i) Execute mapper phase job
Mapper
Reducer
(ii) Perform logical operations according to row key list
Row key list
Larger table
Smaller table
(iii) Store results
HBase

Fig. 6 Implementation of CROSS JOIN

take a larger table as the input of this MapReduce job. Three steps are described as follows.

(i) Each Mapper gets the partial data of a larger table in the HBase as its input.
(ii) The Mapper gets the row key list and takes the combination of row keys of the larger and smaller tables as the new row key. When processing the input row of the larger table, Mapper replaces the row key part of KeyValue instance with the new row key. Mapper then adds this instance to the Put instance. Mapper also has to get the raw data for each row key in the row key list and do the same steps as above described. The Put is an instance provided by the original HBase package used to output to HBase. Finally, we get the Put instance containing the cross join result of the current input row.
(iii) The Mapper sets the Put instance as the value of MapReduce key-value pair. The MapReduce framework outputs the Put instance after Mapper completes the job.

The method for CROSS JOIN queries aims at the cases that one table is much smaller than the other one. Thus, each mapper performs CROSS JOIN with partial data of the large table and the complete data of the small table along with small row key list. While handling both tables with very large size, a very large row key list will be generated first. Each mapper then needs more spaces to perform CROSS JOIN with partial data of the large table and another large table along with the produced large row key list. To avoid performance dropping dramatically, partitioning the other large table is a possible solution. There are fewer researches studies discussing on the cross join than the equal join. The concept of most common solutions of equal join is similar to the partitioning.

(d) *Sorting query*: The sorting query is a combination of the clause with ORDER BY. In HBase, rows of the table are lexicographically sorted by the row key. Since the data stored in HBase is the type of byte array, we regard the row key as characters to compare any two keys from left to right. The row keys in our design are the primary keys of original database tables. To process the sorting query, the row key is composed of the value of the specified column and the row keys of input table.

For example, the column DIE_X of the input table DIE is the specified column to be sorted and the row keys of table DIE is column DIE_SN. The corresponding row key of the result is in the format: "DIE_X:DIE_SN". The column DIE_X is used to be the information for sorting and the column DIE_SN is used to guarantee each row is unique.

5 Experimental results

To evaluate the proposed MapReduce algorithms in JackHare framework for SELECT, FROM, WHERE and extended clauses, we implemented them to compare with Hive and MySQL. We describe the experimental environment and results as follows.

5.1 Experimental environment

To evaluate the performance of the proposed MapReduce algorithms, we established a 16-node virtual machine cluster on four physical machines. The installed version of Hadoop on the cluster is 0.20.203. The version of HBase installed on the top of Hadoop is 0.92.0, which is the datastore of the proposed framework and also supports Hive to access to an existing HBase table with string and byte data types. (Hive HBase Integration (2013). https://cwiki.apache.org/Hive/hbaseintegration.html) To evaluate the performance, Hive was installed on the same cluster. The version of Hive is 0.9.0 which supports creating an external table to access a table in HBase with byte data type. Since Hadoop is a Java-based system, the installed version of JAVA is 1.6.0 and the maximum heap size is 512 MB. A layer-2 switch is used to connect to physical machines. Every physical machine has two Intel Xeon L5640 CPU, 24 GB ram and 3 TB HD. In the 16-node virtual machine cluster, each virtual machine has two cores at 2 GHz with 4 GB ram and 400 GB storage space. The version of MySQL database 5.1.52 and was installed on a virtual machine with two cores at 2 GHz, 4 GB ram and 800 GB hard disk to compare the performance disparity among SQL database and NoSQL database.

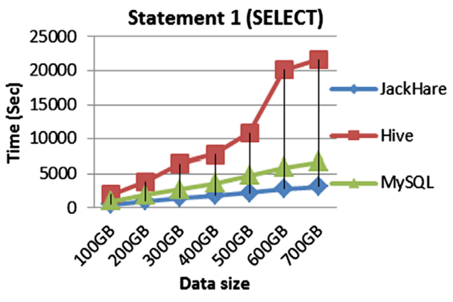
The large-scale dataset for evaluating different approaches was generated from a semiconductor industry. The SQL queries to manipulate data for different approaches are listed in Table 2, and are adopted as benchmarks to evaluate the performance. The dataset is composed of three tables in RDB. The table names are LOT, WAFER and DIE for storing data of lot, wafer and die. One lot has 25 wafers and each wafer has two million dies, the table die is extremely larger than the others. To import the dataset to HBase, three tables are combined in a HBase table with data model described in Sect. 4.1. Hive then creates external table to access data in HBase.

5.2 Results

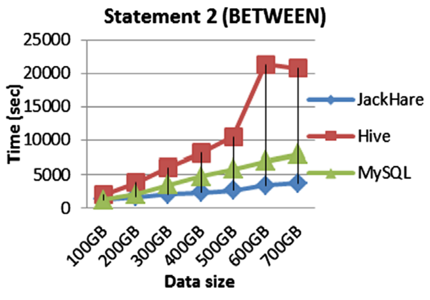
This section classifies the SQL statements into three categories which are "SELECT, FROM, WHERE", "GROUP BY and AGGREGATE FUNCTIONs" and "ORDER BY and JOIN", and presents the performance results as follows.

Table 2 SQL Statements of data manipulations for performance benchmark

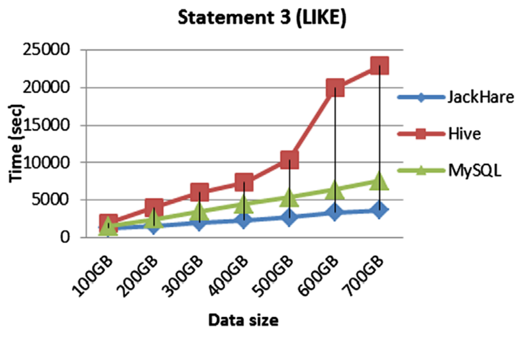
	Statements
1.	SELECT ∗ FROM DIE WHERE WAFER_ID = ’A00002.01’
2.	SELECT ∗ FROM DIE WHERE WAFER_ID BETWEEN ’A00002.20’ AND ’A00002.25’
3.	SELECT ∗ FROM DIE WHERE WAFER_ID LIKE ’A00002.2 %’
4.	SELECT WAFER_ID, COUNT(∗) FROM DIE WHERE WAFER_ID LIKE ’A00002.2 %’ GROUP BY WAFER_ID
5.	SELECT SUM(PSVDDV), MAX(PSVDDV), MIN(PSVDDV), AVG(PSVDDV) FROM DIE WHERE WAFER_ID LIKE ’A00002.2 %’
6.	SELECT WAFER_ID, PSVDDV FROM DIE WHERE WAFER_ID BETWEEN ’A00002.20’ AND ’A00002.25’ ORDER BY WAFER_ID DESC
7.	SELECT ∗ FROM LOT a, WAFER b



(a)



(b)

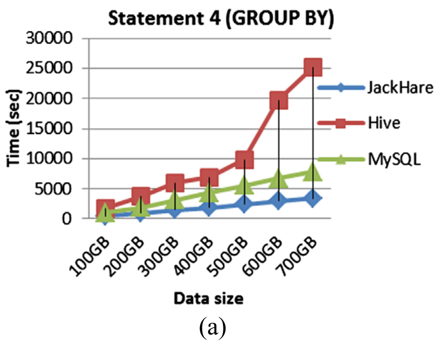


(c)

Fig. 7 Performance of SELECT, FROM, WHERE (**a**) Statement 1 (**b**) Statement 2 (**c**) Statement 3

5.2.1 SELECT, FROM, WHERE

The first category consists of statements 1, 2 and 3 which addresses on filtering the data with different constraints. The experimental results are shown in Fig. 7. In our work, the result of each query is directly stored in the HBase. That is, the number of



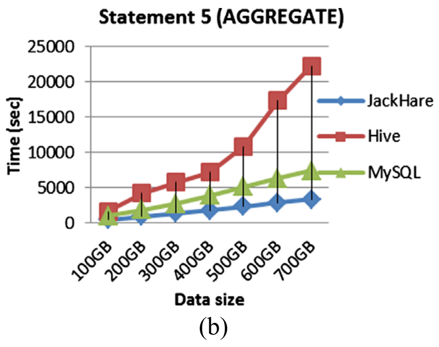


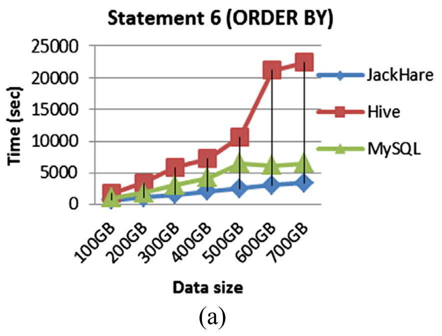
Fig. 8 Performance of GROUP BY and AGGREGATE FUNCTIONs (**a**) Statement 4 (**b**) Statement 5

rows written to the result table may affect the total execution time. In the experience of statement 1, there are 2,000,000 rows in result. In the cases of statement 2 and statement 3, there are 12,000,000 rows in result for each statement. On average, the experimental results show that the statement 2 and the statement 3 spend more time than the statement 1 to process the dataset for JackHare.

The performance gap between JackHare, Hive and MySQL becomes larger as the size of dataset increases. In the case of 100 GB, we find that our work performs almost the same with MySQL in statement 2. That is because the I/O time of HBase takes a great proportion of the execution time. Therefore, when the output is relatively small to the total input dataset, the difference of performance between our method and MySQL is not obvious. In other cases, JackHare has better performance against MySQL. On the other hand, JackHare outperforms Hive in this category. The reason could be the characteristic of Hive descripted in YSmart (Lee et al. 2011) that Hive is a production SQL-to-MapReduce translator. The MapReduce jobs created by Hive may be inefficient. JackHare is 3, 1.5 and 1.5 times faster than Hive with 100 GB data size for statement 1, 2 and 3 respectively. JackHare has better performance with larger data size against Hive, and is almost 7.06 times faster with 700 GB data size and 4.7 times faster on average in this category.

5.2.2 GROUP BY and AGGREGATE FUNCTIONs

While the first category focuses on the queries of I/O part, the second category addresses the performance of computation queries. Figure 8 presents the performance results for this category of statement 4 and statement 5, in which the number of output rows are 6 and 1 respectively. The results depict that the execution time of JackHare is less than the results in the previous category. The phenomenon illustrates that computation intensive queries are more effective than I/O intensive queries for JackHare to process a large-scale data in HBase. Generally speaking, JackHare outperforms Hive and MySQL and is at least 5.04 and 2.15 times faster than Hive and MySQL for statement 4 and 5 on average.



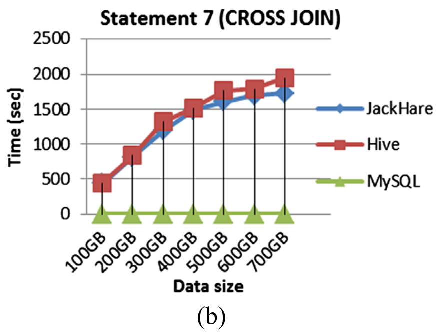


Fig. 9 Performance of ORDER BY and JOIN (**a**) Statement 6 (**b**) Statement 7

5.2.3 *ORDER BY and JOIN*

The experimental results of these two kinds of statements are shown in Fig. 9. The result of statement 6 is similar to previous statements. JackHare outperforms its competitors in all cases. In the statement 6, the number of output row is 12,000,000 according to the WHERE clause. The performance gap between JackHare, Hive and MySQL increases as the size of dataset grows. Comparing statement 6 with statement 2 for JackHare, we can find the ORDER BY query costs less time than the basic query under the same size of dataset. That is because fewer columns are specified in the SELECT clause. Accordingly, the more columns a query needs to output the result, the more execution time will be.

Statement 7 is a JOIN query which queries data from two tables. Since LOT and WAFER tables are small, the relational database performs CROSS JOIN no more than one second. However, the relational database cannot perform CROSS JOIN with an extremely large table such as DIE. While executing CROSS JOIN with LOT and WAFER (two small size of tables), JackHare performs better than Hive in all data sizes. It performs 1.08 times faster than Hive on average and saves 675 seconds in total.

5.2.4 *Summary*

The exact values for graphs in Figs. 7, 8, and 9 are presented in Table 3. In general, the more input data a query needs to deal with, the larger execution time will consume. Nevertheless, in all cases of benchmarked queries, our approaches can resolve each query with stable performance.

6 Conclusions

In this paper, we have proposed the JackHare framework with a comprehensive solution including SQL query compiler, JDBC driver and a systematical method using MapReduce for processing the unstructured data in NoSQL database. JackHare is developed based on Hadoop and HBase to store the data that originally resides in the

Table 3 Exact values for graphs in Figs. 7, 8 and 9

Statement	Method	Results of different data size in second						
		100 GB	200 GB	300 GB	400 GB	500 GB	600 GB	700 GB
Statement 1	JackHare	503	947	1427	1740	2164	2743	3077
	Hive	1962	3790	6437	7730	10907	20109	21701
	MySQL	996	1822	2645	3582	4729	5864	6680
Statement 2	JackHare	1334	1616	2062	2296	2679	3434	3746
	Hive	1961	3810	6053	8165	10522	21323	20721
	MySQL	1265	2087	3452	4701	5792	6985	7934
Statement 3	JackHare	1241	1543	2042	2282	2667	3312	3673
	Hive	1903	3989	5995	7340	10393	19951	22920
	MySQL	1511	2425	3476	4420	5331	6343	7545
Statement 4	JackHare	471	938	1449	1813	2387	2944	3372
	Hive	1790	3707	6002	6935	9857	19770	25233
	MySQL	1026	1859	3124	4292	5490	6692	7811
Statement 5	JackHare	473	961	1461	1832	2340	2889	3376
	Hive	1697	4283	5789	7216	10856	17331	22272
	MySQL	1021	1924	2765	3963	5215	6371	7441
Statement 6	JackHare	667	1132	1597	2045	2532	3058	3400
	Hive	1713	3445	5903	7269	10704	21225	22497
	MySQL	1049	1807	3083	4238	6546.82	6186.6	6482.5
Statement 7	JackHare	439	813	1194	1487	1614	1695	1732
	Hive	448	837	1322	1522	1770	1798	1952
	MySQL	0.01	0.01	0.01	0.01	0.01	0.01	0.01

relational database and designed the corresponding MapReduce methods based on the logic of SQL queries. To ease the learning curve of employing NoSQL database like HBase, SQL user can import the JDBC driver to SQL client software to manipulate the large-scale data without being aware of NoSQL database. The experimental results show that JackHare outperforms both Hive and MySQL in most cases and is also a scalable solution for various sizes of data. However, result of statement 7 shows the difficulty of replacing RDBs while executing CROSS JOIN with small tables. In the future, we will focus on a hybrid architecture combining the best of both RDBs and JackHare to improve JackHare framework.

References

Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A.: HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In: Proceedings of the VLDB Endowment. VLDB Endowment, Armonk pp. 922–933 (2009)

Afrati, F.N., Ullman, J.D.: Optimizing joins in a map-reduce environment. In: Proceedings of the 13th International Conference on Extending Database Technology, pp. 99–110 (2010)

Apache Hadoop: http://hadoop.apache.org (2013)

Apache HBase: http://hbase.apache.org (2013)

Binnig, C., Rehrmann, R., Faerber, F., Riewe, R.: FunSQL: it is time to make SQL functional. In: Proceedings of the 2012 Joint EDBT/ICDT Workshops, pp. 41–46. ACM, New York (2012)

Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A comparison of join algorithms for log processing in MaPreduce. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp. 975–986 (2010)

Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. **26**(2), 1–26 (2008)

Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)

Ghemawat, S., Gobioff, H., Leung, S.: The Google file system. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, pp. 29–43. ACM, New York (2003)

Gowraj, N., Ravi, P.V., Sumalatha, M.R.: S2MART: smart sql to map-reduce translators. In: Proceedings of the Web Technologies and Applications. LNCS, vol. 7808, pp. 571–582. Springer, Berlin (2013)

Hive HBase Integration (2013). https://cwiki.apache.org/Hive/hbaseintegration.html

Kaldewey, T., Shekita, E.J., Tata, S.: Clydesdale: structured data processing on MapReduce. In: Proceedings of the 15th International Conference on Extending Database Technology, pp. 15–25. ACM, New York (2012)

Lee, R., Luo, T., Huai, Y., Wang, F., He, Y., Zhang, X.Y.: Yet another SQL-to-MapReduce translator. In: Proceeding of the 2011 31st International Conference on Distributed Computing Systems, Washington, pp. 25–36 (2011)

Okcan, A., Riedewald, M.: Processing theta-joins using MapReduce. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp. 949–960. ACM, New York (2011)

Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 1099–1110. ACM, New York (2008)

Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pp. 165–178. ACM, New York (2009)

Stonebraker, M., Cattell, R.: 10 rules for scalable performance in ’simple operation’ datastores. Commun. ACM **54**(6), 72–80 (2011)

Su, X., Swart, G.: Oracle in-database hadoop: when mapreduce meets RDBMS. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 779–790. ACM, New York (2012)

Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: a warehousing solution over a map-reduce framework. In: Proceedings of the VLDB Endowment. VLDB Endowment, Armonk pp. 1626–1629 (2009)

Xu, Y., Hu, S.: QMapper: a tool for SQL optimization on hive using query rewriting. In: Proceedings of the 22nd International Conference on World Wide Web Companion, pp. 212–221. ACM, Geneva (2013)