Unified Access Layer with PostgreSQL FDW for Heterogeneous Databases

Abstract. Large-scale application systems usually consist of different databases for different purposes. However, the increasing use of different databases, especially NoSQL databases, makes it increasingly challenging to use and maintain such systems. In this paper, we demonstrate a framework to design a foreign data wrapper (FDW) for external data sources, then we use these FDWs of different databases to build a unified access layer. We propose a novel method to access heterogeneous databases including SQL and NoSQL databases in a unified way. This method was tested in real business applications of Alibaba, in which we were able to do various operations on Redis, MongoDB, HBase, and MySQL by using a simple SQL statement. In addition, the information exchange and data migration between these databases can be done by using unified SQL statements. The experiments show that our system can maintain good database performance and provide users with a lot more convenience and efficiency.

Keywords: Unified access layer, Heterogeneous databases, Foreign data wrapper, HBase, MongoDB.

1 Introduction

In the big data era, traditional relational databases started to seem somewhat powerless in the face of the massive data processing and analysis requirements. To fill this gap in demand, a wide variety of non-relational NoSQL databases were developed. However, the CAP theorem [2] states that it is impossible for a distributed computer system to simultaneously provide all of the following three guarantees: consistency, availability, and partition tolerance. Different NoSQL databases are designed and built according to different feature orientations, which means that their schemas and semantics of data may be significantly different. In a practical project, in order to fully combine the advantages of using a variety of database capabilities, a large-scale system will often integrate a variety of databases, including SQL and NoSQL, to support it. And this also brings some big challenges to the deployment and maintenance. Different databases often stand on their own and have different APIs, with no uniform standard at all. When a new database component is added, the development and maintenance staff need to learn a new set of interfaces, test new drivers, and write extract-transform-load (ETL) [3] components that exchange data with other databases.

Typically, users must interact with these databases at the programming level with customized APIs. This reduces portability and requires system-specific codes. There exist many solutions which aim to integrate data from separated relational systems into a new one. However, most NoSQL systems do not support SQL. Instead, more

adfa, p. 1, 2011.

attention has been paid to their different architectural design decisions in order to achieve better scalability and higher performance. On the other hand, SQL systems can be more portable and expressive, and also have the most trained users. Some commercial companies have recognized this and combined an SQL relational processor with a MapReduce query processor [4]. However, many of the most popular NoSQL databases, such as MongoDB and HBase, do not have SQL interfaces for their systems.

In this paper, we demonstrate a framework to design a foreign data wrapper (FDW) for external data sources, then we use these FDWs of different databases to build a unified access layer. And we propose a method to build a unified access layer for heterogeneous databases based on PostgreSQL, which can translate traditional SQLs into the corresponding NoSQL database APIs to access heterogeneous database components. In our system, the users only need to know SQL to operate all kinds of databases transparently. And our system was tested in real business applications of Alibaba, which convincingly confirms the usefulness of the method.

The main contributions of this paper are as follows:

- We propose a framework to design a framework to design a FDW for external data sources, and we realize FDWs for HBase and MongoDB by using this framework.
- By using a uniform SQL abstraction layer, a database with different capabilities hides the details, which can expose the performance characteristics of the hybrid transactional/analytical processing (HTAP) system at the same time.
- Our work can greatly reduce the complexity of development and maintenance because the staff responsible for development and maintenance can use SQL for all the operations. In our system, ETL components can be achieved by writing SQL, which can be much easier to understand.

2 Related Work

2.1 Heterogeneous databases integration solutions

Generally, data integration methods mostly aim to integrate data arising from different SQL systems. For example, Inspur Company proposed a method to access different types of relational databases[1], but it did not consider the requirements of non-relational databases. However, NoSQL systems play an important role in many domains [10], especially in Big Data domains involving web data, which require supporting millions of interactive users or performing analytics on terabytes of data, such as web logs and click streams. It is very difficult to build a relational database for data in these domains. As a result, a variety of NoSQL databases are created to handle larger data volumes with better performance than relational systems [11]. There is another major advantage of NoSQL databases: they are simpler and more flexible so programmers can use them more easily.

[1] Patent no. CN 101645074 A

To our best knowledge, there are only a few works which aim at standardization for a variety of NoSQL systems and SQL systems. In [6], the Save Our System (SOS) was proposed, which defined a common API for Redis, MongoDB, and HBase. It makes it easy to access through different NoSQL databases, but it cannot handle SQL-based access well and the main drawback of the method is that the expressive power of the implemented methods is limited exclusively to single objects of NoSQL systems. In [8], a relational layer supporting SQL queries and joins was added on top of Amazon SimpleDB. SQL query conversion was done by using regular expressions and rules rather than using a query parser and optimizer. However, it applies only to Simple DB, and other NoSQL databases are not applicable. In fact, ISO/IEC 9075-9:2008 has defined the SQL/MED, or Management of External Data, extension to the SQL standard. We will further discuss SQL/MED in Sect. 3.

2.2 Challenges of heterogeneous databases integration

During information retrieval from heterogeneous source systems we have to face two main challenges: (i) resolving the semantic heterogeneity of data including resolving the structural (data model) heterogeneity of data and (ii) bridging differences of data querying syntaxes.

Semantic heterogeneity means differences in meaning and interpretation of context from the same domain. Semantic heterogeneity in schema level arises from synonyms and homonyms of attributes [38]. A synonym problem occurs when the same real world entity is named differently in different databases, and homonym problem occurs when different real world objects (e.g. entities and attributes) have the same name in different databases [38]. *Structural heterogeneity* (data model heterogeneity) is arising from different modeling approaches and from the use of different data models. On one side, in a given type of data model, real world objects can be modeled in different ways. On the other side, structural heterogeneity is further complicated if data are represented in different data models. NoSQL systems do not implement the traditional relational data model, and according to the implemented type of the data model they can be divided into different types, including key-value stores, document stores, MapReduce systems and graph databases [1]. *Syntactic heterogeneities* have to be faced as well. SQL is a standardized programming language, designed for managing data stored in relational database management systems. However, NoSQL systems are based on different data models, they implement different access methods which are not compatible with the SQL language.

Based on these heterogeneities we can see that data integration is a challenging task. In this paper, we aim at building a unified access method which could hide the specific details and heterogeneities of the various databases.

3 Management of External Data

The prototype system used PostgreSQL and MongoDB and allowed joining relational data with data in MongoDB using the SQL/MED wrapper [5]. However, it is mainly given as a concept rather than to achieve a highly available system.

SQL/MED provides extensions to SQL that define foreign-data wrappers and data-link types to allow SQL to manage external data. External data is defined as data that is accessible to, but not managed by, an SQL-based database management system (DBMS). This standard can be used in the development of federated database systems [13]. There are two current mainstream approaches based on SQL/MED:

Apache Drill is an implementation of SQL/MED. Apache Drill is a distributed system for interactive ad-hoc analysis of large-scale datasets [14]. A single query can join data from multiple databases [14]. In fact, Drill is a JDBC-based implementation of various external data sources and it has been used for some SQL syntax extensions. The main problem of Drill is that it requires independent operations and maintenance support which means that it cannot handle databases well without schema.

PostgreSQL FDW is another implementation of the SQL/MED standard that provides an extension for managing external data sources. There are now a variety of foreign data wrappers (FDWs) available that enable PostgreSQL server to access different remote data stores, ranging from other SQL databases through to flat files [15]. However, there are still a few problems with it: (i) Many NoSQL databases do not have FDWs, such as HBase. (ii) Most of these wrappers are not officially supported by the PostgreSQL Global Development Group (PGDG) and some of these projects are still in Beta version. Even for the FDW given by PGDG, it may still have some mistakes. For example, MongoDB_FDW has some permissions and installing errors.

Based on PostgreSQL FDW, we has done some effective work and tested our system in an actual business system.

4 Overall Framework

In this paper, we carried out our research work based on a business system of Alibaba. The system stores and maintains statistical information on many applications. Each piece of statistical information uses application identification ‘app_id’ and business date ‘date’ as the primary key, including various schema-free statistical indicators. As a result, back-end data storage uses MongoDB and HBase. HBase stores historical data and MongoDB maintains real-time statistical information.

This system needs to provide external queries of indicators and each line of business needs different indicators. Each project team needs to write complex, repetitive code to handle business data queries, including HBase and MongoDB data models, the corresponding driver API, the agreed field format and serialization method, row-key construction logic, and non-transparent sharding logic. From the project development test to the acceptance may take a group 1 or 2 months, which is very costly. In

order to improve efficiency and avoid duplicate work, we built a unified access layer for public data.

By building a unified access layer, some previous problems become easy to solve and many pieces of complex codes can be changed to simple SQL statements. API prototype development takes only a few minutes, greatly improving the efficiency of development and business agility.

Later, databases such as MySQL, Redis, and others have been gradually added through the public access layer, which greatly facilitates the R&D personnel who need to maintain the business logic.

A framework of the system is shown in Fig. 1. In this system, PostgreSQL stores the business metadata, HBase stores historical business data, MongoDB stores real-time business data, and Redis is used as the cache. Based on PostgreSQL a unified access layer is built and makes all the other databases SQL or NoSQL transparent for users. By this way, users could use SQL to operate all these databases.

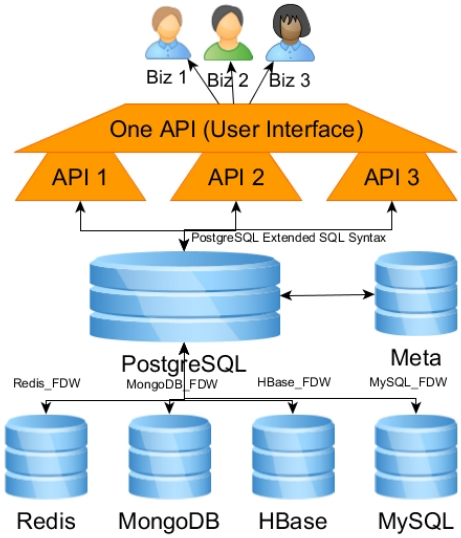

Fig. 1. Framework of the unified access layer system

5 Design FDW for External Databases

Based on the SQL/MED method, we used PostgreSQL FDW as a platform to manage external data sources. In this section, we propose a design framework of FDW for external databases. And as examples, we will introduce the design of the FDWs for HBase and MongoDB following this framework.

5.1 Design Framework of FDW

As Fig. 2 shows, to design an FDW, we need firstly analyze the target API set of an external databases. Since different databases always have different API set especially for NoSQL-databases. For example, *find ()* is used to realize the query function in MongoDB database but *get ()* or *scan ()* may be used to do the same query in HBase database. Hence only by knowing the API set of an external database could we know which operations need to be added corresponding in FDW. Secondly, we have to design different SQL syntax for different data storage methods. Unlike relational databases, NoSQL databases tend to have their own unique data storage methods. So we need to design corresponding SQL syntax for data definition language (DDL) of external databases. Finally, we consider how to execute a complex query with a variety of constraints without much performance loss, and we called this process as conditional pushdown.

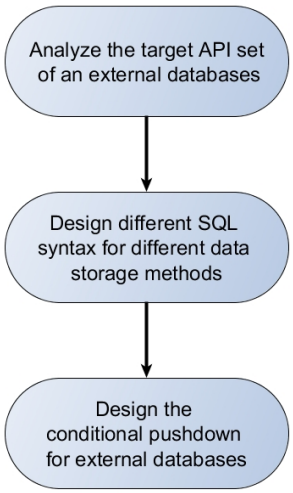

Fig. 2. Design framework of FDW

Following this framework, we design two FDWs for HBase and MongoDB in a business system, called HBase_FDW and MongoDB_FDW respectively.

5.2 Design of HBase_FDW

As a popular database, it is a pity that no FDW implementation for HBase is available publicly. Maybe that is because HBase is a part of Hadoop's ecosphere and it is quite difficult to access HBase via non-Java languages. The HBase Protobuf interface is only available after Version 0.96.

In this section, we firstly analyze the data model and API of HBase, then we design the SQL grammar and the conditional pushdown of it. Our work is open-source and you can access it on GitHub[2].

Data Model and API

HBase is a Bigtable-like distributed database. It is a sparse long-term storage (on the hard disk), multi-dimensional, sorted mapping table [16]. The index of this table is the rowkey, the column family, and the timestamp. The data type of HBase is string, with no other type. The user stores the data in the table, with each row having a sortable primary key and any number of columns. Because it is sparse storage, different rows of the same table may have different numbers of columns. In short, the HBase data model is a large table, the properties of which can be dynamically increased according to the needs, but there is no relational query between the tables.

The HBase API consists: get, scan, put, and delete.

SQL Grammar

First we need to design a grammar for querying HBase.

FDWs manage foreign data as 'relations' (i.e., a table) so it is important to organize foreign data into tables. We establish an abstraction for HBase according to the physical data model of HBase with the following quintuple:

(rowkey, family, qualifier, timestamp, value)

We can index to a unique value so this quintuple could be a candidate. However, such a schema design does not meet the usual data usage. The SQL syntax for data definition language (DDL) of external tables requires more elaborate forms of expression.

Then we design the fields in external tables. Firstly, because rowkey is the core design of HBase, it is necessary to include rowkey in the DDL. Secondly, for the column families and columns that we are interested in, we should put them as fields in the DDL of the external tables. Because the distinction between column family and column is not significant for the users, and column families are more likely to exist as a physical storage-optimized hint message, we can put the column family and column together, combined into the DDL column. Since the same family of data has the same access pattern, it is also a good way to put column families in external table names or additional parameters in external tables.

Because the data type of HBase is string, for the external table field type, from the data model point of view, rowkey, column family, column, and value are stored in a byte array, which does not contain any type information. Thus, the most common standard method is to set the type of these fields as byte array, which is called BYTEA in PostgreSQL.

In accordance with the above consideration, a sample DDL for an external data is shown in Table 1.

---

[2] https://github.com/Vonng/hbase_fdw

Table 1. A sample DDL for an external data

App_user_stat (table name in HBase)	
Column name	**Type**
rowkey	BYTEA
active	BYTEA
install	BYTEA
launch	BYTEA

Because all field types are byte array, the user needs to manually convert the query. As Figure 2 shows, for the string users need to follow the encoding and decoding, and for other data types, they need to process data according to different serialization programs in the query statement. In this way, users can execute SQL operations on data that they are interested in.

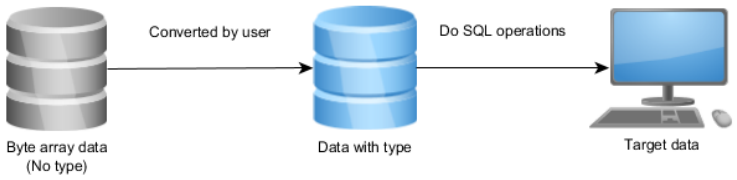

Fig. 3. The process of using byte array data

HBase itself has no type information, but through PostgreSQL DDL as Table 2 shows, we can add external data sources with the type information, which will make reading and using more convenient.

Table 2. A sample DDL through PostgreSQL

App_user_stat (table name in HBase)		
Column name	**Type**	**Options**
rowkey	TEXT	
timestamp	TIMESTAMP	
active	INTERGER	qualifier '1_day_active_count'
install	INTERGER	qualifier '1_day_install_count'
launch	INTERGER	qualifier '1_day_launch_count'

In this way, we can directly apply a variety of conditions in the SQL query and easily filter the data.

```
SELECT rowkey,active,install,launch FROM app_user_stat

WHERE rowkey BETWEEN '9c9a' AND '9c9c' AND active > 0

 AND install > 0 AND rowkey ~ '^.{4}_.{10}_\w{24}' LIMIT
10;
```

The above-mentioned SQL generated the following execution plan in PostgreSQL:

```
Limit (cost=20.00..5020.00 rows=10 width=52)

 -> Foreign Scan on app_user_stat
(cost=20.00..50000000.00 rows=100000 width=52)

    Filter: ((rowkey >= '9c9a'::text) AND (rowkey <=
'9c9c'::text) AND (active > 0) AND (install > 0) AND
(rowkey ~ '^.{4}_.{10}_\w{24}'::text))
```

At the time of execution it will be HBase_FDW, translated as:

```
hbase.scan('app_user_stat',

 TScan(startRow='9c9a', stopRow='9c9c',

  columns=[TColumn('stat', '1_day_active_count'),

      TColumn('stat', '1_day_launch_count'),

      TColumn('stat', '1_day_install_count')],

  filterString="RowFilter(=, 'regex-
string:^.{4}_.{10}_\w{24}')"))
```

To achieve this goal, we need to complete the core issues of the FDW design: conditional pushdown.

Conditional Pushdown

A practical system must meet the efficiency requirements. For FDW, the most critical step is conditional pushdown, which means translating the 'where' clause in an SQL query statement to the corresponding external database API operation. Without conditional pushdown, each query will be a full table scan, and Postgres has to carry out conditional filtering after all the data is fetched to Postgres, which means the whole system will be very inefficient with large-scale data. By adding conditional pushdown in FDW, we could minimize performance overhead in the system as much as possible.

For HBase, we have done the following work on its different operations[3]:

1. Unconditional query: we push it down to a full table scan.

$$\sigma = \text{scan}$$

```
SELECT * FROM hbase.appuserstat_table;
```

2. Query with a rowkey: we push it down to the GET operation.

[3] The first line of each condition is HBase API, and the second line is the corresponding SQL statement.

$$\sigma_{key = k} = get(k)$$

```
SELECT * FROM hbase.appuserstat_table WHERE rowkey=k;
```

3. Query with rowkey's size comparison condition: we push it down to SCAN's startRow and stopRow.

$$\sigma_{k1<key<k2} = scan(startRow = k1, stopRow = k2)$$

```
SELECT * FROM hbase.appuserstat_table WHERE rowkey
BETWEEN k1 AND k2;
```

4. Query with an 'in' expression: we push it down to getMultiple operation.

$$\sigma_{key\ belongs\ to\ \{k1,k2,...,kn\}} = getMultiple([k1,k2,\ldots,kn])$$

```
SELECT * FROM hbase.appuserstat_table WHERE rowkey in
(k1,k2,…, kn);
```

5. Query with the Regex or 'like' expression of a given rowkey: we push it down to the combination of SCAN and RowFilter String.

$$\sigma_{key \sim regexp} = scan(filter = \text{“ RowFilter(='regexstring : regexp')”})$$

```
SELECT * FROM hbase.appuserstat_table where name REGEXP
regexp;
```

6. Query with the list of columns required: we push it down to column filter.

$$\sigma_{column\ belongs\ to\ \{c1,c2,...,cn\}} = scan(column = [c1,c2,\ldots,cn])$$

```
SELECT * FROM hbase.appuserstat_table WHERE column in
(c1,c2,…, cn);
```

The above mentioned are all the fundamental operations of HBase. Any complex operations can be split into a combination of these basic operations, so that can be translated into the corresponding SQL statement.

5.3 Design of Mongo_FDW

For MongoDB, the PGDG has provided an FDW[4], Mongo_FDW, designed for a relatively complete set of rules, and mapping the SQL syntax to the MongoDB API. The following describes how Mongo_FDW works.

Data model

MongoDB is a document-based NoSQL database, where Collection corresponds to the concept of Relation in the relational database, and a Document corresponds to a

[4] https://github.com/EnterpriseDB/mongo_fdw

Record [17]. The biggest difference between MongoDB and a relational database is that Collection is schema-free, so each Document can optionally contain different fields in MongoDB. Because SQL itself is used to query the structured data, when we need to use the SQL syntax to access MongoDB some schemas are necessary for Collection. It is very useful that the free expansion of the field can get very good support through the JSON type of PostgreSQL.

SQL grammar

In HBase we need to design Rowkey to put the application ID and date together. In MongoDB, app_id and date are independent. The fields specified in the foreign table DDL are reflected in the Projection parameter in the MongoDB API.

Conditional Pushdown

The conditional pushdown in Mongo_FDW is essentially a WHERE clause that describes the Filter object of MongoDB. The Filter of MongoDB is represented by a method called SON Manipulator[5]. Table 3 shows the correspondence in SON Manipulator. The main work of Mongo_FDW is translating SQL statements to objects in SON.

Table 3. The correspondence in SON Manipulator

SQL	SON	SQL	SON
=	$eq	<	$lt
>	$gt	In	$in
>=	$gte	Not in	$nin
<=	$lte	- regexp	$regex
<>	$ne		

On the basis of the above work, we started to build up a unified access layer.

6 Unified access layer via PostgreSQL

In this section, we will give a practical example to illustrate how to build up a unified access layer and to demonstrate the great convenience it brings.

We created two tables as shown in Table 4, in which a) corresponds to the historical data stored in HBase and b) is the real-time data stored in MongoDB. The table consists of rowkey, active, install, and launch information.

Table 4. a) Foreign table of historical data

Appuserstat_history (table name in HBase)		
Column Name	**TYPE**	**OPTIONS**
rowkey	TEXT	

[5] http://api.mongodb.com/python/current/api/pymongo/son_manipulator.html

active	INTEGER	qualifier '1_day_active_count'
install	INTEGER	qualifier '1_day_install_count'
launch	INTEGER	qualifier '1_day_launch_count'

b) Foreign table of real-time data

Appuserstat_real-time (table name in MongoDB)	
Column Name	**TYPE**
app_id	NAME
date	DATE
active	INTEGER
install	INTEGER
launch	INTEGER

After building the external table, an SQL statement will be translated into the API of HBase or MongoDB and return the result.

Through these two DDL, we can see the difference between these two models. The ideal interface should hide this difference. That is, when the date is today, the request will be automatically routed to MongoDB and when the date is before today, the request will be routed to HBase accordingly. We use stored procedures in PostgreSQL to complete the combination of real-time indicators and historical indicators, which is so-called Lambda architecture.

Through this unified access layer, users can only use this simple form to query the application of statistical data:

```
-- When query today's index, Route to MongoDB

SELECT * FROM appuser-
stat_merge('56444370e7e12af0561e221c', CURRENT_DATE);

-- When query historic index, Route to HBase

SELECT * FROM appuser-
stat_merge('56444370e7e12af0561e221c', '2016-02-02');
```

Any PostgreSQL driver, shell, or graphical user interface (GUI) can perform such a query. This is just an example of a stored procedure to handle the construction logic of rowkey and routing logic for real-time data and historical data. Some other functions and logics, like sharding or routine, could also be done in this way.

In this section, we illustrated how the system works. However, it is only a part of our system. That is, the unified access layer not only support HBase and MongoDB but also supports Redis and MySQL, as Figure 1 shows. Moreover, our system has strong scalability. If it is needed, more kinds of databases can be added.

7 Evaluation

For large-scale heterogeneous databases, the main costs involve the operation of the database itself and the FDW only takes up a small part of the overhead. Thus the use

of FDWs does not have much impact on the efficiency of the system. At the same time, there are many advantages of using FDWs, including accelerating the development progress and making the system easier to maintain.

In order to test the performance of the system, we deployed two different systems and used the same hardware conditions, which are Intel Dual-core and 2G memory. One of them uses HBase as the single database with the original operation of HBase, and the other one uses our system to access HBase through unified access layer. After 10,000,000 queries on the test, the comparison of these two systems in terms of query per second (QPS) and request delay is shown in Table 5.

Table 5. Comparison between the two systems

Type of system	CRUD QPS	Request delay
HBase only	164.89	6.0465 ms
Through unified access layer	164.85	6.066 ms

Through the above comparative tests, we can see that using our system will not significantly affect the efficiency.

And we have conducted statistical studies on the development of multiple projects, Table 6 shows the comparison between system with unified access layer and previous system. By using the unified access layer in UMeng of Alibaba Group. The original few months of the workload can be done by only one person in just a few days. At the same time, users can readily use Hbase, even without much information about the data model or API of HBase. Whether the work is development, debugging, testing, real-time monitoring, or ETL, many complex logics can be completed through SQL.

Table 6.

Project name	Development cycle	
	With unified access layer	No unified access layer
Project A		

8 Conclusion

With the arrival of the big data era, a large system often consists of a variety of different databases. Different databases have different data models and operations. To make the systems easier to use, people have tried to find some ways to use heterogeneous databases with a unified method. In this paper, based on a practical system, we designed and implemented a unified access layer for heterogeneous databases.

We applied the FDW technology to our production practice in order to solve practical problems and we proposed HBase FDW to fill gaps in related fields. With only SQL, we can perform operations on all the databases in this system and the experimental results showed that the efficiency of our system was satisfactory. As more and more advanced databases are being created and used, understanding how these databases are used becomes time-consuming and laborious. We anticipate that in the future unified access will become a trend and FDW technology will be widely used.

References

1. Patel J M. Operational NoSQL Systems: What's New and What's Next? [J]. Computer, 2016, 49(4):23-30.

2. E Brewer, “A certain freedom: thoughts on the CAP theorem[C],” Proceedings of the 29th ACM SIGACT-SIGOPS symposium on principles of distributed computing. ACM, 2010, pp. 335–335.
3. P Vassiliadis, “A survey of Extract–transform–Load technology [J],” International Journal of Data Warehousing and Mining (IJDWM), 5(3), 2009, pp. 1–27.
4. H Yang, A Dasdan, R L Hsiao, et al., “Map-reduce-merge: simplified relational data processing on large clusters[C],” Proceedings of the 2007 ACM SIGMOD international conference on management of data, ACM, 2007, pp. 1029–1040.
5. R Lawrence, “Integration and virtualization of relational SQL and NoSQL systems including MySQL and MongoDB[C],” Computational Science and Computational Intelligence (CSCI), 2014 International Conference on IEEE, 1, 2014, pp. 285–290.
6. P Atzeni, F Bugiotti, L Rossi, “Uniform access to non-relational database systems: The SOS platform[C],” International Conference on Advanced Information Systems Engineering, Springer Berlin Heidelberg, 2012, pp. 160–174.
7. R Vilaça, F Cruz, J Pereira, et al., “An effective scalable SQL engine for NoSQL databases[C],” IFIP International Conference on Distributed Applications and Interoperable Systems, Springer Berlin Heidelberg, 2013, pp. 155–168.
8. A Calil, R dos Santos Mello. “SimpleSQL: a relational layer for SimpleDB[C],” East European Conference on Advances in Databases and Information Systems, Springer Berlin Heidelberg, 2012, pp. 99–110.
9. J Tatemura, O Po, W P Hsiung, et al., “Partiqle: An elastic SQL engine over key-value stores[C],” Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, ACM, 2012, pp. 629–632.
10. A Botta, W De Donato, V Persico, et al. “Integration of cloud computing and internet of things: a survey [J],” Future Generation Computer Systems, 56, 2016, pp. 684–700.
11. S Venkatraman, K Fahd, S Kaspi, et al., “SQL Versus NoSQL Movement with Big Data Analytics [J],” 2016.
12. J Roijackers, G H L Fletcher, “On bridging relational and document-centric data stores[C],” British National Conference on Databases, Springer Berlin Heidelberg, 2013, pp. 135–148.
13. J Melton, J E Michels, V Josifovski, et al., “SQL/MED: a status report [J],” ACM SIGMOD Record, 31(3), 2002, pp. 81–89.
14. M Hausenblas, J Nadeau, “Apache drill: interactive ad-hoc analysis at scale [J],” Big Data, 1(2), 2013, pp. 100–104.
15. I Ahmed, A Fayyaz, A Shahzad, “PostgreSQL Developer's Guide [M],” Packt Publishing Ltd, 2015.
16. L George, “HBase: The Definitive Guide: Random Access to Your Planet-Size Data [M],” O’Reilly Media, Inc., 2011.
17. K Chodorow, “MongoDB: The definitive guide [M],” O’Reilly Media, Inc., 2013.