2011 International Conference on Transportation, Mechanical, and Electrical Engineering (TMEE)
December 16-18, Changchun, China

Middleware Design for Integrating Relational Database and NOSQL Based on Data Dictionary

Hailing Zhang, Yang Wang, Junhui Han
Dept. of Electronic and Information Engineering,
Harbin Institute of Technology Shenzhen Graduate School
Shenzhen, China
zhhailg@163.com, wangyang@hitsz.edu.cn, hanjh404@sina.com

***Abstract*—With the rapid development of WEB2.0 or the Social Web as well as the maturing of Internet retail, the traditional relational database is increasingly incompatible with the demands of availability, scalability for massive data, fast data backup and recovery. With the original purpose of large-scale web applications, NOSQL concept, abbreviation of "not only structured query language", is proposed, which is a broad class of database management systems that differ from classic relational database management systems. NOSQL databases generally do not require a fixed formal schema and usually avoids join operations, so its efficiency for simple query is very high but not for complex query. In this article, a method is proposed to integrate these two types of database by adding middleware between application layer and database layer. We mainly design the detailed implementation of two modules which are data dictionary module and SQL processor module of middleware. The middleware would provide an unified interface and shield underlying data distribution for the applications, meet the need of horizontal scalability without affecting the application logic implementation.**

Keywords-NOSQL; database integration; middleware; data dictionary;

I. INTRODUCTION

NOSQL was coined in 1998 by Carlos Strozzi to describe this lightweight, open-source database that did not expose a SQL interface. Eric Evans reintroduced the term NOSQL in early 2009 when Johan Oskarsson of Last.fm wanted to organize an event to discuss open-source distributed databases. Because Internet applications weaken demands for relational model, while have strong demands for large-scale data storage, NOSQL is prompted greatly and is popularized by large web sites such as Google, Facebook and Digg. Amazon supplies an open-source version named Dynamo[1], which is a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. Google are using Bigtable in many products and projects, including Google Earth, Google Finance, Pesonalized Search, Google Analytics[2]. Although the products are different on demanding workloads, Bigtable supplies a flexible and high-performance solution successfully, which range from through-oriented batch-processing jobs to latency-sensitive serving of data to end users. MongoDB[3] is document (JSON style) based data storage engine, whose company announced formally not long ago that MongoDB support Windows platform. Numerous internal portals and the emerging WEB2.0 website also use open source technology of NOSQL to solve practical problems.

Although some radical tissues or persons claim that relational database is dying, but we know clearly that relational database is alive very well, in some area, it is indispensable, because it can provide an unparalleled feature set, and also play perfectly on the data integrity and stability. That is also a fact, not all of the features are necessary. Relational model artificially distort the nature contents of the object, its data model is too rigid and strict, especially not conducive to the operate WEB 2.0 applications' request.

In practical, it is application developers' job to decide whether to use relational database or NOSQL, in the scene requiring storing large amounts of data, they select NOSQL to meet data storage needs. However, most NOSQL systems employ a distributed architecture, with the data held in a redundant manner on several servers, and partitioning scheme relies on consistent hashing to distribute the load across multiple storage hosts[4], as a result, query processing and indexing efficiency are not high enough, and complex query is difficult to complete. Therefore, relational database can be used in conjunction with NOSQL since it handles relational structure proficiently. In Memcached plus MySQL architecture, Memcached[5] is used as an independent distributed cache server. In this article, we design NOSQL as distributed storage instead of cache.

This paper presents integration of relational databases and NOSQL approach by appending middleware, which provides an unified interface processing across various databases. Applications do not have to consider the underlying data model, storage location and semantic heterogeneity. And this model fulfills scalability of the massive data without affecting application logical implementation. Middleware can integrate relational database, and many other types of distributed storage, such as Cassandra[6], Hbase[7], Redis so on. Each NOSQL product has specific storage model and features of its own; you can choose one based on business scenarios.

The rest of the paper is structured as follows. Section II presents the integration architecture. Section III describes the detailed implementation of middleware. Section IV

978-1-4577-1701-7/11/$26.00 ©2011 IEEE

analyzes our system and gives the performance testing result. Section V concludes the paper.

II. DATABASE INTEGRATION ARCHITECTURE

The architecture integrating relational database and NOSQL architecture is shown in Figure 1. It includes functional parts as follow: middleware client, middleware server, and SQL engine.

1) Middleware client

Middleware client is integrated into application, is responsible for the communication between application and middleware server, including user/password encryption.

2) Middleware server

Middleware server is a distributed data transmission service, shielding distributed and heterogeneous data sources which bring inconveniences to the application cluster, achieves transparent access to data. Middleware server accepts request from application, analyzes it then transfers the request to the underlying database. This article designs data dictionary, SQL processor and result set handler in detail.

3) SQL engine

NOSQL products generally do not provide a standard SQL interface; instead, they access their data with the help of internal API. SQL engine converts standard SQL requests into various types of access to the NOSQL database. It acts as wrapper integrated into NOSQL cluster's each nodes.

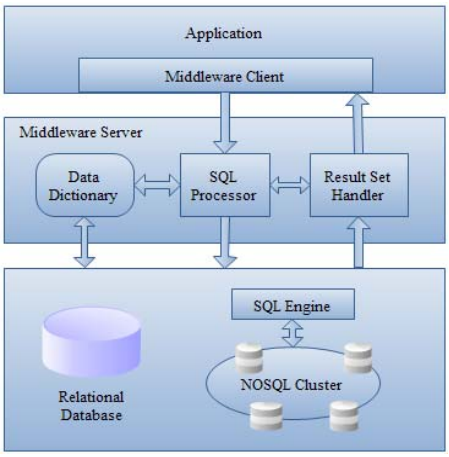

Figure 1. Database integration architecture.

III. KEY TECHNOLOGIES TO BE DESIGNED

A. Data Dictionary Design

Data dictionary takes essential part of the architecture for completing analysis of SQL requests and dispatching tasks. In order to communicate with various types of databases, data dictionary holds the routing information as well as node information describing each database source.

We use table name to determine which type of database to be accessed, so business developer must ensure that the various databases in the same domain can not establish the same name table, or else, the routing information on the basis of the table name will come into confusion.

In this article, we choose Cassandra as a kind of NOSQL product. With the identifier "rowkey" to be the routing field, its column family is equivalent to table of relational database. We list several tables of the most important ones in data dictionary below: 1)VersionCtrl(the table for controlling version); 2)RouteForDBType(the routing table for database type); 3)RouteForRDB(the routing table for relational database); 4)RouteForCass(the routing table for Cassandra); 5)DBInfo(the table of database nodes information).

Middleware maintains the routing model as shown in Figure 2.

B. SQL Processor and Result Set Handler Design

Steps of traditional SQL Engine's job are: 1) Input: input SQL request such as insert, delete, update, select or call procedure and so on. 2) Parse SQL: parse the whole SQL statement; convert it into internal data structure. 3) SQL Verify: check the validity of tables and fields appeared, using schema data dictionary. 4) Optimize: decide the table joining order and table scan mode, index will affect the plan. 5) Execute: execute the SQL request. 6) Output: output the result from database server.

This article borrows ideas from traditional SQL Engine's procedure to design our SQL Processor module. Firstly, we abandon SQL Verify, Optimize, Execute, which will be done by database layer when DB executes SQL request. Secondly, routing parse is added in. According to routing data dictionary introduced in section A, SQL request is divided into several sub-requests matching their database type. Also, Result Set Handler Module is newly added to integrate the results returned from multiple data sources. What's more, SQL Processor only needs to parse part of the whole SQL statement in the case that the SQL statement involves to access unique data source, this design will save lots of time for parsing. SQL Processor's flow chart is shown in Figure 3.

We give an example; the following request is only related to access NOSQL:

- select name, age from table1 where id = "AAAAAAA" and addr = "beijing" order by age group by name ;

We only need to parse until table name, that is table1, then access the table naming RouteForDBType described in data dictionary, have cleared out table1 is stored in NOSQL, following where condition we just care of routing field, in this example, it is "id", the rest statement including group by, order by, etc. does not need to be analyzed, this whole SQL request is sent to NOSQL implementation then.

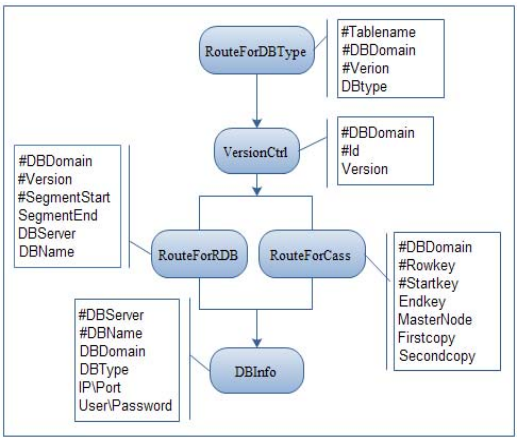

Figure 2. Data dictionary design model

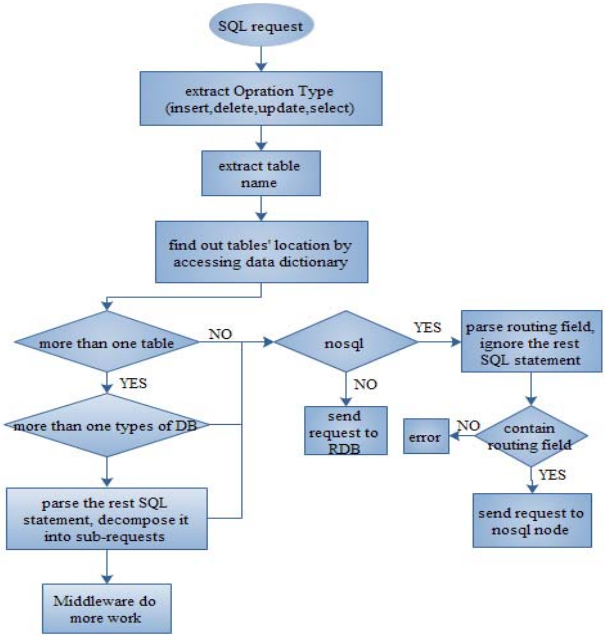

Figure 3. SQL Processor's flow chart

Result Set Handler Module directly passes the result set from database to application in form of stream without integration and analysis having understood that application accesses only one type of database.

If application SQL request accesses NOSQL as well as relational DB, middleware will play more work. This article proposes a solution to rapid integration of heterogeneous database. There are two cases to access to multiple database, we will discuss respectively below.

1) Middleware accesses databases concurrently

To obtain an object's information, application usually provides the primary key, which is used to access NOSQL as a routing field. In this case, middleware will send each sub-request concurrently to databases; this will improve utmost response speed.

As shown in Figure 3, before SQL Processor decomposes the request into sub-request, we have obtained several lists in step 1 and step 2.

Step 1: OperationType_list: Include but not just the following operation type: insert, delete, update and select.

Step 2: From_list: It is a collection of tables appeared in the SQL request.

Next, SQL Processor parses whole statement, following job will be done:

Step 3: Extract required fields to build Field_list as table to be a unit.

Step 4: Extract conditional expression in condition statement after the keyword "where", build where_list. If the expression deals with two data sources' field, we put it into an extraordinary list Relation_list.

Step 5: According to what lists in 1), 2), 3), 4), we restructure SQL statement, the formation of a number of sub-requests. Send the sub-requests to their databases at the same time, cache the result set returned from each DB.

Step 6: Result Set Handler Module filter result set been cached. According to Relation_list, we determine which result is needed.

Step 7: At last, middleware returns the final result to the application.

This article will give another example to show how middleware to work when SQL request is going to access both types of DB as following:

- select table1.name table2.major from table1, table2 where table1.id = "AAAAAA" and table1.age = table2.age and table2.addr = "beijing";

  Processing steps are shown in Figure 4.

2) Middleware accesses databases one by one

For range query, application usually does not specify primary keys, brings on traversal of all NOSQL nodes, which makes very low query efficiency. To solve this problem, we send conditional filter to relational database at first to obtain the qualified primary keys, with which we then access NOSQL. The detailed steps are similar to what we discuss in section *1)*, except step 5, which send sub-request to database concurrently.

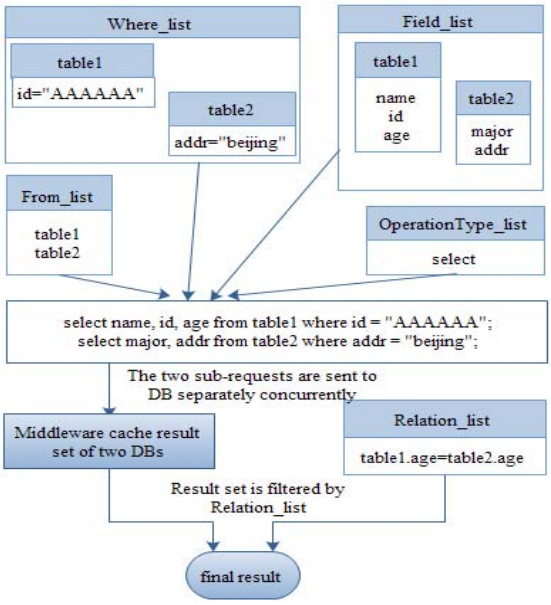

Figure 4. Middleware decomposes SQL request and handles result sets

IV. ANALYZE THE SYSTEM

A. Detailed feature analysis

We analyze our architecture, and illustrate the functions and conveniences contrast between before and after using middleware module in Table I. App is the abbreviation of application.

B. Test result of performance

Along with the functions and conveniences of our system, a little more costs will be brought compared with no using the middleware module. We did an experiment to evaluate the performance loss. Before adding middleware, application connects database directly, and maintains database metadata, designs data distribution rule, also, parses routing information before executing SQL request. In the experiment, client access one node Oracle database and one node Cassandra service. In figure 5 and figure 6, we illustrate throughput and request time how to change with the number of client connections. The throughput will increase along with client connections and rich to top when client connections are 200. Throughput will decrease about 2000 records per second in average and request time will

TABLE I. CONTRAST OF BEFORE AND AFTER MIDDLEWARE

Compare items	Before middleware	After middleware
Apps configure DB connections with all the database nodes	Yes	No, only configure connections with a few middleware nodes
Alter all apps' configurations when add a DB server	Yes	No, only alter the middleware's configuration
Apps apply different DB drivers when access different types of databases	Yes	No, only apply an unify JDBC driver
Apps design data distribution and routing rule, parse SQL request	Yes	No
DB connections sharing	No, apps control a connection pool; real connections can not be shared between different apps. When increase the scale of apps, connections become bottleneck.	Yes, apps' connection pool establishes virtual connection, the real DB connection between middleware and DB is established when needed and released after that.
Scalability	Good	Bad

increase 5 milliseconds for each access. The performance reduction is acceptable for application and middleware's advantages are much more attractive.

Test system is a distributed data service realized by a company, whose name is not suitable to be mentioned. Client accesses database server by using JDBC driver, we choose to test SELECT operation because traditional database serves mostly "read heavy" or "read and write heavy" and NOSQL takes advantage in "write heavy". Then we do the experiment to compare relational database only system and our system. Each SELECT operation's interval is four milliseconds. The query table contains thirty fields; each field has a fixed length of twenty bytes.

The database storage node is Tecal RH2285 server, detailed parameters are as follows: a)CPU: Intel(R) Xeon(R) CPU X5570@2.93GHz 8 Processors; b)Memory:4GB; c)Disk: 4 TB; d)Bandwidth: 1Gbps.

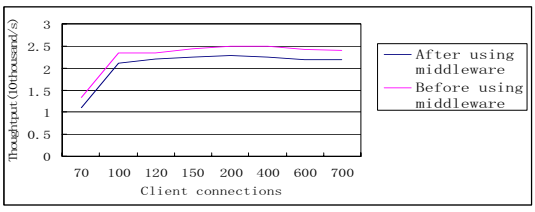

Figure 5. Throughput comparison

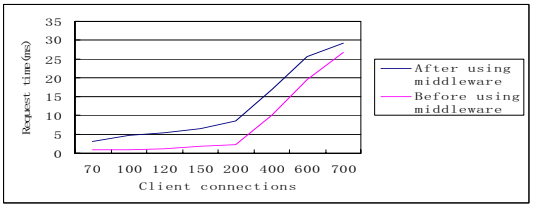

Figure 6. Request time comparison

V. CONCLUSION

For database integration, the biggest obstacle is the heterogeneity and distribution of data. Without affecting the original system, building middleware that can achieve rapid integration of existing database systems. Middleware provides application layer exactly an uniform interface and shields the underlying data sources' heterogeneity and distribution, to achieve transparent access to data. Application only needs to focus on the logical development without having to understand the underlying data distribution and structure.

Although NOSQL has the advantage of horizontal expansion, but for complex SQL requests, it can not support them very well. For the query based on KEY/VALUE and massive data storage requirements, NOSQL is a good choice[8]. In this case, the combination of relational databases and NOSQL will bring many benefits.

REFERENCES

[1] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubra-manian, Peter Vosshall and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, page 205-220, New York, NY, USA, 2007. ACM.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghenmawat, Wilson C. Hsieh, Deborah A. Wallach Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In proceeding of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation – Volume 7 (Seattle, WA, November 06 -08, 2006). USENIX Association, Berkeley, CA, 15-15.

[3] Kristina Chodorow, Michael Dirolf. MongoDB: The Definitive Guide.

[4] Karger D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In Proceedings of the Twenty-Ninth Annual ACM Symposium on theory of Computing (EI Paso, Texas, United States, May 04 – 06, 1997). STOC '97. ACM Press, New York, NY, 654-663

[5] Khairina Abu Bakar, Shaharill, Mohd Hafiz Md, Ahmed, Mohiuddin. Performance evaluation of a clustered mencache. Information and Communication Technology for the Muslim World (ICT4M). 2010, E54 – E60.

[6] Avinash Lakshman and Prashant Malik. Cassandra – a decentralized structured storage system. Technical report, Cornell University, 2009.

[7] Brussels, Belgium. Supporting Multi-row Distributed Transactions with Global Snapshot Isolation Using Bare-bones HBase. The 11th ACM/IEEE International Conference on Grid Computing (Grid 2010), Oct 25-29, 2010, Brussels, Belgium.

[8] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Cauthier, P. 1997. Cluster-based scalable network services. In Proceeding of the 16th ACM Symposium on Operating Systems Principles (Bolton Landing, NY, USA, October 19 – 22, 2003). SOSP '03. ACM Press, New York, NY, 29-43.