Toward Lightweight Transparent Data Middleware in Support of Document Stores

Kun Ma*, Ajith Abraham† ‡
*Shandong Provincial Key Laboratory of Network Based Intelligent Computing
University of Jinan, Jinan, China
ise_mak@ujn.edu.cn
†Machine Intelligence Research Labs, Scientific Network for Innovation and Research Excellence, WA, USA
‡IT4Innovations, VSB-Technical University of Ostrava, Czech Republic
ajith.abraham@ieee.org

***Abstract*—With the advent of rapid increase in the size of data in legacy applications, it is urgent to build efficient and flexible data middleware that supports SQL to MapReduce transformation. In this paper, we propose a data middleware to translate the SQLs to the operations that the NoSQL can understand using the MapReduce framework we design. This middleware transforms the read and write SQL statements into MapReduce jobs. In addition, a set of transformation rules is discussed in detail. This middleware can significantly reduce redundant computations and improve the performance of I/O operations.**

***Keywords*-MapReduce; big data; middleware; NoSQL; Cloud Computing**

I. Introduction

Traditional relational Database Management Systems (RDBMSs) have been used for decades to store relational data. However, it meets the I/O bottleneck issues in the case of big data, especially for the query operation and full-text searching [1]. To address this limitation, several systems have already emerged to propose NoSQL, standing for 'non-relational' or 'Not only SQL' [2]. It is often used as the storage of the schema-free data to enhance the performance of applications that exhibit a high read to write ratio. Typically, document store (DS), is a kind of popular NoSQL database [3]. It stores document-oriented data in the form of BSON. Some examples of document stores are MongoDB [4] as well as CouchDB [5].

Although NoSQL databases have some advantageous features for specific applications, it is difficult to migrate the legacy applications to support NoSQL. On one hand, it is a huge project to make the legacy applications to support NoSQL. On the other hand, it is impossible to discard the RDBMS completely [6]. Therefore, an eclectic solution is benefiting the application with the integration of RDBMS and NoSQL. In this paper, we focus on this challenge to design a lightweight data middleware to make the legacy applications support document stores transparently.

MapReduce is a distributed programming model, which is an associated implementation for processing and generating big data [7]. First, a map function processes a key/value pair to generate a set of intermediate key/value pairs. Second, a reduce function merges all intermediate values associated with the same intermediate key. The significance of this model is that many real world tasks are expressible in this model. In our paper, we consider the query and transaction operation on the NoSQL as the MapReduce task. In order to improve the performance of the NoSQL data middleware, we design it using MapReduce frameworks.

The rest of the paper is organized as follows. The related work is discussed in Section II. In Section III, we design the architecture of lightweight transparent data middleware in support of document stores. We introduce a SQL parser to intercept the SQLs from the JDBC proxy, and send them to the MapReduce framework to translate them into the operations that the NoSQL can understand. Moreover, the transformation rules from SELECT/INSERT/UPDATE/DELETE statements to the MapReduce tasks are discussed in detail. Brief conclusions are outlined in the last section.

II. Background and Related Work

A. Document Stores

Document stores, known as document-oriented database, is one of the main categories of NoSQL databases. In contrast to well-known RDBMS, these systems are designed around an abstract notion of a document to eschew the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas [3]. Although documents inside a document-oriented database are similar to the records in relational databases, they are not required to adhere to a standard schema, known as schema-free. The basic item of the document is the key/value pair in broader terms.

There are some main features of the document store. First, it supports effective search by field, range queries as well as regular expression searches using indexing, especially in the case of big data. Second, it can run over multiple servers, balancing the load and/or duplicating data to keep the system up and running in case of hardware failure. Third, it can be used for batch processing of data and aggregation operations.

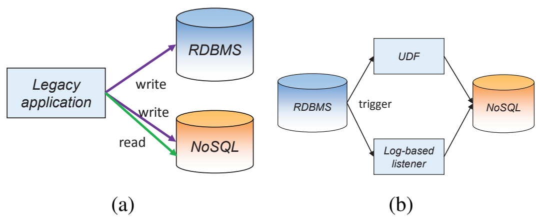


Figure 1. RDBMS first and NoSQL second.

B. MapReduce Framework

MapReduce is a programming model for parallel and distributed computing, which was proposed by Google [7]. The model is inspired by the map and reduce functions commonly used in functional programming, although their purpose in the MapReduce framework is not the same as their original forms. The process is divided into two steps: map and reduce. Developers specify map functions which take an input pair and produces a set of intermediate key/-value pairs. The MapReduce library then groups together all intermediate values associated with the same intermediate keys, and passes them to the reduce functions which are also specified by the programmers; The reduce function accepts an intermediate key and a set of values for that key. It merges together these values to form a possibly smaller set of values. The intermediate values are supplied to the user's reduce function via an iterator. The programming model of MapReduce framework is shown as follows:

- map: $(k_1, v_1) \rightarrow list(k_2, v_2)$. Map Function. Map function takes an input key-value pair (k_1, v_1) and produces a set of intermediate key-value pairs (k_2, v_2).
- reduce: $(k_2, list(v_2)) \rightarrow list(k_3, v_3)$. Reduce Function. Reduce function merges the intermediate key-value pairs $(k_2, list(v_2))$ together to produce a smaller set of values $list(v_3)$.

C. Integration of RDBMS and NoSQL

Currently, there are two categories of methods to benefit from both RDBMS and NoSQL. The first approach is RDBMS first and NoSQL second. In this solution, the legacy applications are rectified to write the data to the RDBMS and NoSQL ate the same time. For the query operation, we can obtain the data from the NoSQL. For the transaction operation, we must write the copy to the NoSQL. The architecture of this solution is shown in Figure 1(a). An improved solution is trigger-based rather than real-time synchronization, which is shown in Figure 1(b). In this solution, the user-defined function or log-based event is designed to trigger the real-time synchronization. The disadvantage is the performance impact on the RDBMS with the mixture of RDBMS and NoSQL.

The second approach is NoSQL first and RDBMS second. In this solution, the legacy applications are rectified to adapt to NoSQL databases. The data are saved in the NoSQL directly. Next, the replication script will synchronize the data to the RDBMS, data warehouse and full-text searching storage. This solution is shown in Figure 2.

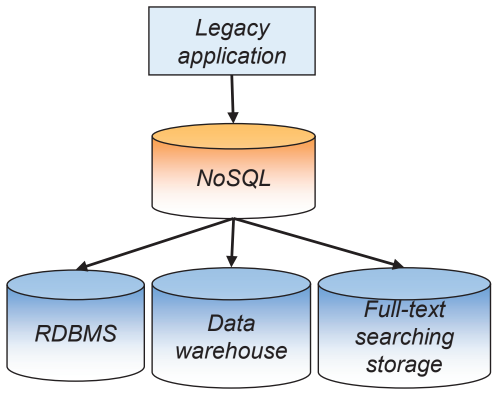


Figure 2. NoSQL first and RDBMS second.

After evaluating the pros and cons of these two approaches, we decided to pursue the last solution. The difference between the NoSQL first solution and ours is that we design a lightweight transparent data middleware to enable the feature of NoSQL without modifying the legacy applications. In the rest of the paper, we introduce the architecture of this middleware. The main difficulty to construct the data middleware is how to convert the SQL-like operations to the similar operations of NoSQL.

D. SQL to MapReduce

This drastic increase in the amount of data has led to the development of extensive data processing middleware like the relational SQL to schema-free query language translations. Currently, there are some translators in practice. Due to the ever increasing size of data, the basic requirement of this middleware is that it can process big data. The obvious approach is parsing the SQL first, and translate it into the corresponding language of NoSQL [8]. The drawback is the inefficiency in case of big data. Another approach is translating the SQL using MapReduce framework [9] [10].

The popularity of a hybrid system or middleware transforming SQL to MapReduce jobs has increased due to the emergence of big data in many data centric organizations. We use MongoDB as the NoSQL storage, and intercept all the SQLs from the JDBC and translate them into the tasks of MapReduce. In this paper, we focus on the translation of the query (select) and transaction operation (insert/update/delete) statements. Currently, there are some translator products. The first is Pig [11], which is developed to support parallelization of data retrieval and analyzing in the Cloud environment. Pig supports a simple language called Pig Latin to support query formulations and data manipulation, which is written using Java class libraries. The second is Hive [12], which is developed by Facebook to add

SQL-like functions to MapReduce. It converts the SQL to HiveQL of using Apache Hadoop. The third is Scope [13], which is developed by Microsoft to generate a new scripting language for managing and retrieving data from large data repositories.

III. Data Middleware

A. Architecture

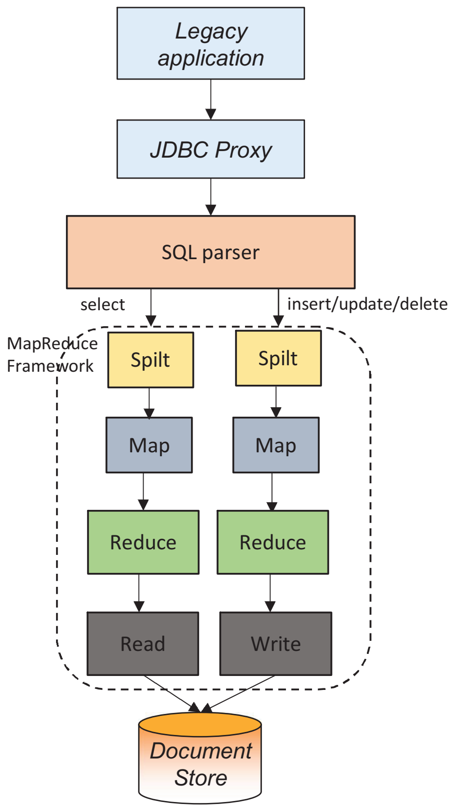


Figure 3. Architecture of NoSQL Data Middleware.

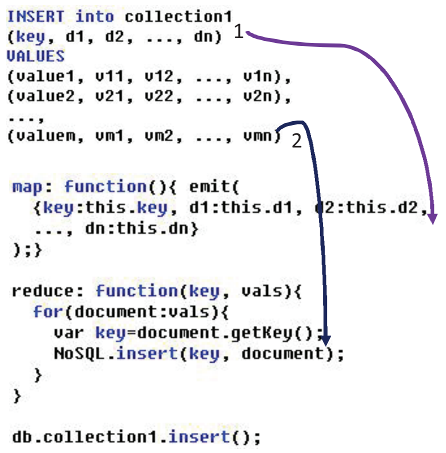
The typical architecture of lightweight transparent data middleware in support of document stores is shown in Figure 3. The architecture is composed of five parts in order to reduce the complexity. From the top to the bottom, they are legacy application, JDBC proxy, select/insert/update/delete parser, MapReduce framework, and document store respectively.

We consider that the legacy application is developed using Java language. We change the original JDBC connection parameters to JDBC proxy, which intercepts the SQL statements. Then the SQL parser forwards the SQLs to MapReduce framework to translate them into the similar operation that NoSQL can comprehend. Next we introduce read/write to MapReduce modules, which are the cores of the MapReduce framework we design.

B. Read to MapReduce

Figure 4 shows the transformation rules from SELECT SQL statement to MapReduce operation on the document store. As shown in Figure 4, the first rule is pulling dimension columns to the map function, reducing the size of the working set. The aggregated functions (e.g. SUM and COUNT) are translated into the operations in the reduce functions, which is described in the second and third rules. The fourth rule is changing the AVG aggregate function into the record count waiting until finalization. The fifth rule is changing query filters into the JSON-like queries. The sixth rule enables the aggregate filtering applied to the result set. The seventh rule is ascending or descending on the query result.

C. Write to MapReduce



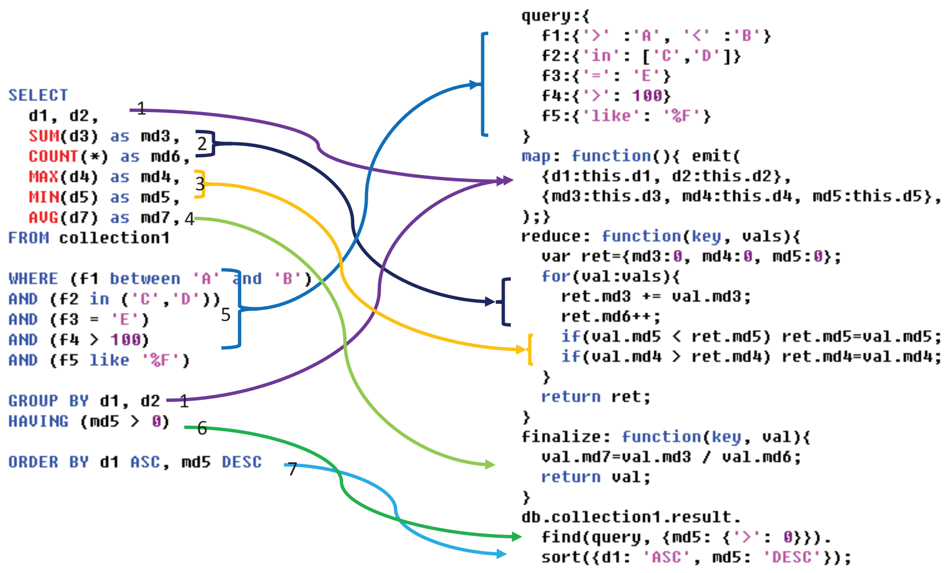
```
INSERT into collection1
(key, d1, d2, ..., dn)
VALUES
(value1, v11, v12, ..., v1n),
(value2, v21, v22, ..., v2n),
...,
(valuem, vm1, vm2, ..., vmn)

 map: function(){ emit(
   {key:this.key, d1:this.d1, d2:this.d2,
   ..., dn:this.dn}
 );}

 reduce: function(key, vals){
   for(document:vals){
     var key=document.getKey();
     NoSQL.insert(key, document);
   }
 }

 db.collection1.insert();
```

Figure 5. INSERT to MapReduce.

Figure 5 shows the transformation rules from INSERT SQL statement to MapReduce operation on the document store. The first rule is pulling dimension columns to the operations in the map function. The second rule is translating the row-based records into the documents in the reduce function.

Figure 6 shows the transformation rules from UPDATE SQL statement to MapReduce operation on the document store. The first rule is pulling dimension columns to the operations in the map function. In the reduce function, the affected documents are saved in the NoSQL. The second rule is changing query filters into the JSON-like queries.

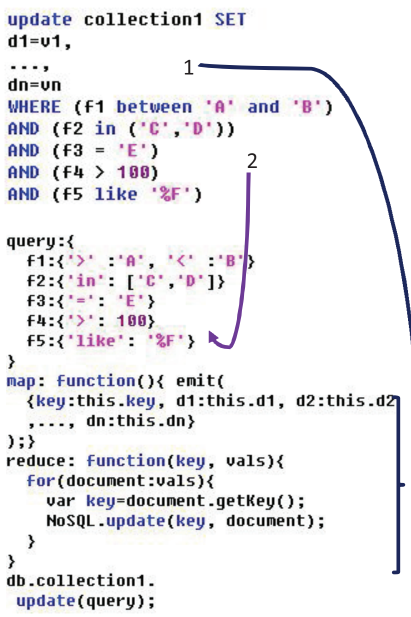
Figure 7 shows the transformation rules from DELETE SQL statement to MapReduce operation on the NoSQL. The first rule is changing query filters into the JSON-like queries. Within the reduce function, the affected documents are deleted.



```
SELECT
  d1, d2,
  SUM(d3) as md3,
  COUNT(*) as md6,
  MAX(d4) as md4,
  MIN(d5) as md5,
  AVG(d7) as md7,
FROM collection1

WHERE (f1 between 'A' and 'B')
AND (f2 in ('C','D'))
AND (f3 = 'E')
AND (f4 > 100)
AND (f5 like '%F')

GROUP BY d1, d2
HAVING (md5 > 0)

ORDER BY d1 ASC, md5 DESC
```

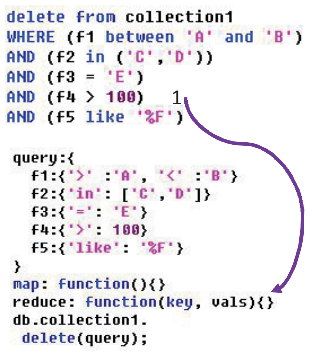
```
query:{
  f1:{'>' :'A', '<' :'B'}
  f2:{'in': ['C','D']}
  f3:{'=': 'E'}
  f4:{'>': 100}
  f5:{'like': '%F'}
}
map: function(){ emit(
  {d1:this.d1, d2:this.d2},
  {md3:this.d3, md4:this.d4, md5:this.d5},
);}
reduce: function(key, vals){
  var ret={md3:0, md4:0, md5:0};
  for(val:vals){
    ret.md3 += val.md3;
    ret.md6++;
    if(val.md5 < ret.md5) ret.md5=val.md5;
    if(val.md4 > ret.md4) ret.md4=val.md4;
  }
  return ret;
}
finalize: function(key, val){
  val.md7=val.md3 / val.md6;
  return val;
}
db.collection1.result.
  find(query, {md5: {'>': 0}}).
  sort({d1: 'ASC', md5: 'DESC'});
```

Figure 4. SELECT to MapReduce.



```
update collection1 SET
d1=v1,
...,
dn=vn
WHERE (f1 between 'A' and 'B')
AND (f2 in ('C','D'))
AND (f3 = 'E')
AND (f4 > 100)
AND (f5 like '%F')

query:{
  f1:{'>' :'A', '<' :'B'}
  f2:{'in': ['C','D']}
  f3:{'=': 'E'}
  f4:{'>': 100}
  f5:{'like': '%F'}
}
map: function(){ emit(
  {key:this.key, d1:this.d1, d2:this.d2
  ,..., dn:this.dn}
);}
reduce: function(key, vals){
  for(document:vals){
    var key=document.getKey();
    NoSQL.update(key, document);
  }
}
db.collection1.
 update(query);
```

Figure 6. UPDATE to MapReduce.



```
delete from collection1
WHERE (f1 between 'A' and 'B')
AND (f2 in ('C','D'))
AND (f3 = 'E')
AND (f4 > 100)
AND (f5 like '%F')

query:{
  f1:{'>' :'A', '<' :'B'}
  f2:{'in': ['C','D']}
  f3:{'=': 'E'}
  f4:{'>': 100}
  f5:{'like': '%F'}
}
map: function(){}
reduce: function(key, vals){}
db.collection1.
 delete(query);
```

Figure 7. DELETE to MapReduce.

IV. Conclusions

Execution of complex queries and transaction operations with high efficiency and high performance is critically desirable for big data legacy applications. In this scenario, the legacy applications encounter the I/O bottleneck issues. Our solution is that we propose a lightweight data middleware in support of document stores, to translate the SQL statements to the NoSQL operations using the MapReduce framework we design transparently. This middleware will be integrated with the other legacy applications without modifying the source codes.

Acknowledgment

This work was supported by the Doctoral Fund of University of Jinan (XBS1237). Ajith Abraham acknowledges the support from IT4Innovations Centre of Excellence project, reg. no. CZ.1.05/1.1.00/02.0070 funded by Structural Funds of the European Union and state budget of the Czech Republic.

References

[1] R. Cattell, "Scalable sql and nosql data stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2010.

[2] N. Leavitt, "Will nosql databases live up to their promise?" *IEEE Computer*, vol. 43, no. 2, pp. 12–14, 2010.

[3] J. Han, E. Haihong, G. Le, and J. Du, “Survey on nosql database,” in *Proceedings of 6th International Conference on Pervasive Computing and Applications*, 2011, pp. 363–366.

[4] E. Dede, M. Govindaraju, D. Gunter, R. S. Canon, and L. Ramakrishnan, “Performance evaluation of a mongodb and hadoop platform for scientific data analysis,” in *Proceedings of of the 4th ACM workshop on Scientific cloud computing*, 2013, pp. 13–20.

[5] G. Manyama, M. A. Paytona, J. A. Rothc, L. V. Abruzzob, and K. R. Coombesa, “Relax with couchdb ł into the non-relational dbms era of bioinformatics,” *ACM SIGMOD Record*, vol. 100, no. 1, p. 1C7, 2012.

[6] S. Khatchadourian, M. Consens, and J. Simeon, “Web data processing on the cloud,” in *Proceedings of 35th SIGMOD international conference on Management of data*, 2009, pp. 165–178.

[7] R. Cattell, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] K. Ma, Z. Chen, A. Abraham, and R. Sun, “A transparent data middleware in support of multi-tenancy,” in *Proceedings of of 7th International Conference on Next Generation Web Services Practices*, 2011, pp. 11–19.

[9] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang, “Ysmart: Yet another sql-to-mapreduce translator,” in *Proceedings of 2011 31st International Conference on Distributed Computing Systems*, 2011, pp. 25–36.

[10] N. Gowraj, P. V. Ravi, M. V, and M. R. Sumalatha, “S2mart: Smart sql to map-reduce translators,” in *Proceedings of 2013 15th Asia-Pacific Web Conference*, 2013, pp. 571–582.

[11] Z. Zhang, L. Cherkasova, A. Vermam, and B. Loo, “Optimizing completion time and resource provisioning of pig programs,” in *Proceedings of 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2012, pp. 811–816.

[12] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, “Hiveca petabyte scale data warehouse using hadoop,” in *Proceedings of 2010 IEEE 26th International Conference on Data Engineering*, 2010, pp. 996–1005.

[13] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, “Scope: Easy and efficient parallel processing of massive data sets,” in *Proceedings of the VLDB Endowment*, 2008, pp. 1265–1276.