Migration from RDBMS to Column-Oriented NoSQL: Lessons Learned and Open Problems

Ho-Jun Kim, Eun-Jeong Ko, Young-Ho Jeon, and Ki-Hoon Lee(✉)

School of Computer and Information Engineering,
Kwangwoon University, 20 Kwangwoon-ro, Nowon-gu,
Seoul 01897, Republic of Korea
kihoonlee@kw.ac.kr

Abstract. Migration from RDBMS to NoSQL has become an important topic in a big data era. This paper provides a comprehensive study on important issues in the migration from RDBMS to NoSQL. We discuss the challenges faced in translating SQL queries; the effect of denormalization, secondary indexes, and join algorithms; and open problems. We focus on a column-oriented NoSQL, HBase, because it is widely used by many Internet enterprises such as Facebook, Twitter, and LinkedIn. Because HBase does not support SQL, we use Apache Phoenix as an SQL layer on top of HBase. Experimental results using TPC-H show that column-level denormalization with atomicity significantly improves query performance, the use of secondary indexes on foreign keys is not as effective as in RDBMSs, and the query optimizer of Phoenix is not very sophisticated. Important open problems are supporting complex SQL queries, automatic index selection, and optimizing SQL queries for NoSQL.

Keywords: Migration · RDBMS · NoSQL · HBase · Phoenix · Denormalization · Secondary index · Query optimization

1 Introduction

NoSQL databases have become a popular alternative to traditional relational databases due to the capability of handling big data, and the demand on the migration from RDBMS to NoSQL is growing rapidly [1]. Because NoSQL has different data and query model comparing with RDBMS, the migration is a challenging research problem. For example, NoSQL does not provide sufficient support for SQL queries, join operations, and ACID transactions.

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2015R 1C 1A 1A02036517).

© Springer Nature Singapore Pte Ltd. 2018
W. Lee et al. (eds.), *Proceedings of the 7th International Conference on Emerging Databases*, Lecture Notes in Electrical Engineering 461,
https://doi.org/10.1007/978-981-10-6520-0_3

In this paper, we provide a comprehensive study on important issues in the migration from RDBMS to NoSQL. We make three main contributions. First, we investigate the challenges faced in translating SQL queries for NoSQL. Second, we evaluate the effect of denormalization, secondary indexes, and join algorithms on query performance of NoSQL. Third, we identify open problems and future work. We focus on HBase because it is widely used by many Internet enterprises such as Facebook, Twitter, and LinkedIn. Because HBase does not support SQL, we use Apache Phoenix as an SQL layer on top of HBase.

Experimental results using TPC-H show that column-level denormalization with atomicity significantly improves query performance, the use of secondary indexes on foreign keys is not as effective as in RDBMSs, and the query optimizer of Phoenix is not very sophisticated. Important open problems are supporting complex SQL queries, automatic index selection, and optimizing SQL queries for NoSQL.

The remainder of this paper is organized as follows. Section 2 presents background and related work. Section 3 discusses important issues in the migration from RDBMS to column-oriented NoSQL. Section 4 presents experimental results on the issues and open problems. Section 5 provides conclusions.

2 Background and Related Work

HBase is a column-oriented NoSQL and uses Hadoop Distributed File System (HDFS) as underlying storage for providing data replication and fault tolerance. HBase does not support SQL queries and secondary indexes. Apache Phoenix works as an SQL layer for HBase by compiling SQL queries into HBase native calls and supports secondary indexes.

Reference [1] proposed a denormalization method called CLDA that avoids join operations and supports atomicity using the notions of column-level denormalization and atomic aggregates. The CLDA method improves query performance with less space compared with table-level denormalization methods [2–8], which duplicate whole tables. For a column-oriented NoSQL, [9] proposed a column partitioning algorithm. Reference [10] studied the implementation of secondary indexes for HBase.

3 Migration from RDBMS to Column-Oriented NoSQL

In this section, we provide a comprehensive study on important issues in the migration from RDBMS to HBase with Phoenix. The issues are exemplified and discussed using a case study on TPC-H.

3.1 Translating SQL Queries

Phoenix does not provide sufficient support for complex SQL queries with complex predicates, subqueries, and views. To migrate such complex queries, we need to simplify complex queries using query unnesting techniques [11–14] and temporary tables.

For example, benchmark queries of TPC-H are very complex, and Phoenix does not sufficiently support queries Q11, Q15, Q18, Q19, and Q21. For Q11, we unnest the subquery in the HAVING clause because Phoenix does not support it. For Q15, we store the result of a view into a temporary table because Phoenix supports only a view defined over a single table using a SELECT * statement. For Q18, we unnest the subquery with the GROUP BY and HAVING clauses because Phoenix produces wrong results. For Q19, Phoenix does not efficiently evaluate a complex predicate of the disjunctive normal form, which is a disjunction of multiple condition clauses. For the query, Phoenix does not push down predicates. To efficiently evaluate the query, we compute results for each condition clause and union the results using temporary tables. For Q21, we unnest the subqueries because Phoenix does not support non-equi correlated-subquery conditions.

3.2 Denormalization

Because NoSQL systems do not efficiently support join operations, we need denormalization, which duplicates data so that one can retrieve data from a single table without joining multiple tables. To denormalize relational schema, we use the method called *Column-Level Denormalization with Atomicity* (*CLDA*) [1], which is the state-of-the-art denormalization method. Although CLDA was originally proposed for a document-oriented NoSQL, it is general enough to be applied to other types of NoSQL. CLDA avoids join operations without denormalizing entire tables by duplicating only columns that are accessed in non-primary-foreign-key-join predicates. CLDA also combines tables that are modified within the same transaction into a unit of atomic updates to support atomicity.

For example, Fig. 1 shows TPC-H Q8 where non-primary-foreign-key-join predicates are shaded. If we add `r_name` to `orders` and `p_type` to `lineitem`, we can avoid "`orders` ⋈ `customer` ⋈ `nation` ⋈ `region`" and "`lineitem` ⋈ `part`." Table 1 shows the columns duplicated by CLDA for the 22 TPC-H queries. The name of each column contains the names of the foreign keys. The number of duplicated columns is small because there are common columns appearing in multiple non-primary-foreign-key-join predicates. According to the TPC-H specifications, the `lineitem` and `orders` tables should be modified within the same transaction. To support transaction-like behavior, CLDA combines the `lineitem` and `orders` tables into a single table. Thus, we can avoid "`orders` ⋈ `lineitem`" with atomicity.

```
select
    o_year,
    sum(case
            when nation = 'BRAZIL' then volume
            else 0
    end) / sum(volume) as mkt_share
from
    (
        select
            extract(year from o_orderdate) as o_year,
            l_extendedprice * (1 - l_discount) as volume,
            n2.n_name as nation
        from
            part, supplier, lineitem, orders, customer,
            nation n1,nation n2, region
        where
            p_partkey = l_partkey
            and s_suppkey = l_suppkey
            and l_orderkey = o_orderkey
            and o_custkey = c_custkey
            and c_nationkey = n1.n_nationkey
            and n1.n_regionkey = r_regionkey
            and r_name = 'AMERICA'
            and s_nationkey = n2.n_nationkey
            and o_orderdate between
                date '1995-01-01' and date '1996-12-31'
            and p_type = 'ECONOMY ANODIZED STEEL'
    ) as all_nations
group by
    o_year
order by
    o_year;
```

Fig. 1. TPC-H Q8.

Table 1. Columns duplicated by the CLDA method for the 22 TPC-H queries.

Table	Duplicated columns
supplier	s_nationkey_n_name
partsupp	ps_partkey_p_brand ps_partkey_p_type ps_partkey_p_size ps_suppkey_s_nationkey_n_name ps_suppkey_s_nationkey_n_regoinkey_r_name
orders	o_custkey_c_nationkey o_custkey_c_mktsegment o_custkey_c_nationkey_n_name o_custkey_c_nationkey_n_regoinkey_r_name
lineitem	l_partkey_p_name l_partkey_p_brand l_partkey_p_type l_partkey_p_size l_partkey_p_container l_suppkey_s_nationkey l_suppkey_s_nationkey_n_name

3.3 Secondary Indexes

Phoenix offers a secondary index on top of HBase using an index table, which consists of index columns and the primary key of the indexed data table. The query optimizer of Phoenix internally rewrites the query to use the index table if it is estimated to be beneficial. If the index table does not contain all the columns referenced in the query, Phoenix accesses the data table to retrieve the columns not in the index table. Phoenix also offers a covered index, which is an index that contains all the columns referenced in the query. Using a covered index, we can avoid the costly access to the data table, but the overhead of data synchronization and space consumption increase.

3.4 Join Algorithms

Phoenix supports a sort-merge join and a broadcast hash join. The broadcast hash join first computes the result for the expression at the right-hand side of a join condition and then broadcasts the result onto all the cluster nodes; each cluster node has a partition of the table at the left-hand side and computes the join locally. When both sides of the join are bigger than the available memory size, the sort-merge join should be used. Currently, the query optimizer of Phoenix does not make this determination by itself. We can force the optimizer to use a sort-merge join by using the USE_SORT_MERGE_JOIN hint.

4 Experimental Evaluation

4.1 Experimental Setup

For the migration from RDBMS to HBase with Phoenix, we evaluate the effect of denormalization, secondary indexes, and join algorithms on query performance. Using the TPC-H benchmark with scale factors (SFs) 1 and 10, we measure the average query execution time for the TPC-H queries. For each query, we first run the query once to warm up the cache and then measure the average execution time for two subsequent runs.

We use HBase 0.9.22, Phoenix 4.8.1, and MySQL 5.7.18. All experiments were conducted on a cluster of four PCs with an Intel Core i5-6600 CPU, 16 GB of memory, Samsung 850 PRO 256 GB SSDs, and Ubuntu 16.04. We set the JVM memory to 12 GB. One PC is a master, and the other three PCs are slaves. For MySQL, we use only one PC.

We conduct the following experiments.

Experiment 1: The effect of denormalization
To see the effect of denormalization, we compare query performance for the denormalized schema generated by the CLDA method and for the normalized schema, which has a one-to-one correspondence with the relational schema. We also compare database size. We use secondary indexes on foreign keys and the USE_SORT_MERGE_JOIN hint for all the queries.

Experiment 2: The effect of secondary indexes on foreign keys
To see the effect of secondary indexes on foreign keys, we compare query performance for databases with and without secondary indexes on foreign keys. We use the normalized schema and the USE_SORT_MERGE_JOIN hint for all the queries. We also run the same test for MySQL to see the effect of secondary indexes on RDBMS.

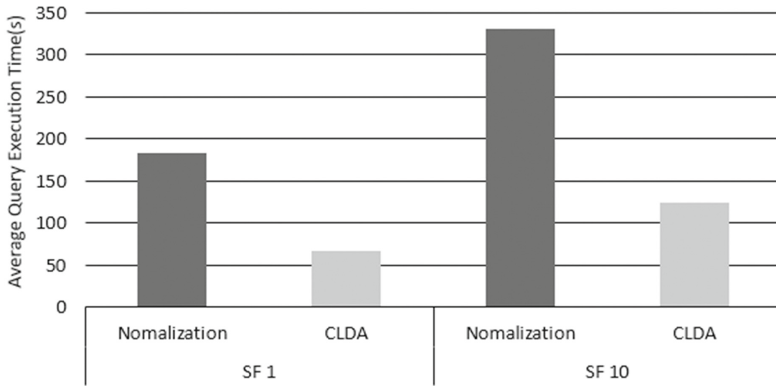
Experiment 3: The effect of join algorithms

To see the effect of join algorithms, we compare query performance with and without the USE_SORT_MERGE_JOIN hint. We exclude queries that are failed due to out-of-memory errors if the USE_SORT_MERGE_JOIN hint is not used. We use foreign key indexes and the normalized schema.

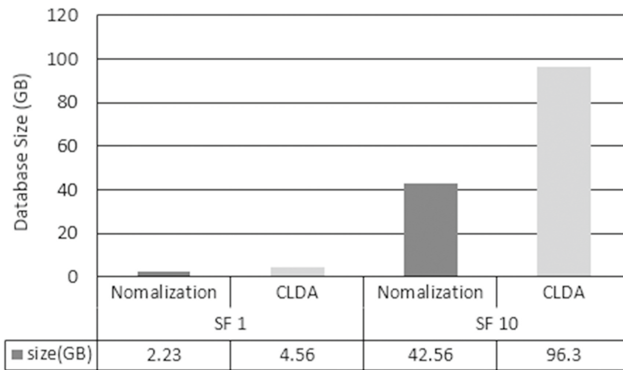
4.2 Experimental Results

Experiment 1: The effect of denormalization

Figure 2 shows that the CLDA method significantly improves query performance at the expense of using more space compared with the normalization method that uses the



(a) Query performance with/without CLDA



(b) Database size with/without CLDA

Fig. 2. The effect of column-level denormalization with atomicity

relational schema as it is. This is because the CLDA method reduces the number of joins by duplicating columns. For SF 1, the CLDA method is 2.7 times faster, but uses 2.0 times more space. For SF 10, the CLDA method is 2.7 times faster, but uses 2.3 times more space. We note that, for SF 10, queries Q2, Q7, Q8, Q9, Q17, and Q21 failed for the normalization method; queries Q9, Q13, Q17, Q18, and Q21 failed for the CLDA method. Queries Q2, Q7, Q8, Q9, Q13, and Q18 failed due to out-of-memory errors; queries Q17 and Q21 failed due to HRegionServer failures. We exclude the failed queries.

Experiment 2: The effect of secondary indexes on foreign keys
For MySQL, secondary indexes on foreign keys are very effective. Without secondary indexes, 73% of queries (16 queries) takes more than one hour, and the average query execution time of the other 27% (6 queries) is 5.4 s even for SF 1. With secondary indexes, the average query execution time of all queries is 0.2 s for SF 1. Figure 3 shows that for HBase with Phoenix, the average query execution times with and without secondary indexes are almost the same for both SFs 1 and 10. For SF 10, we exclude the failed queries.

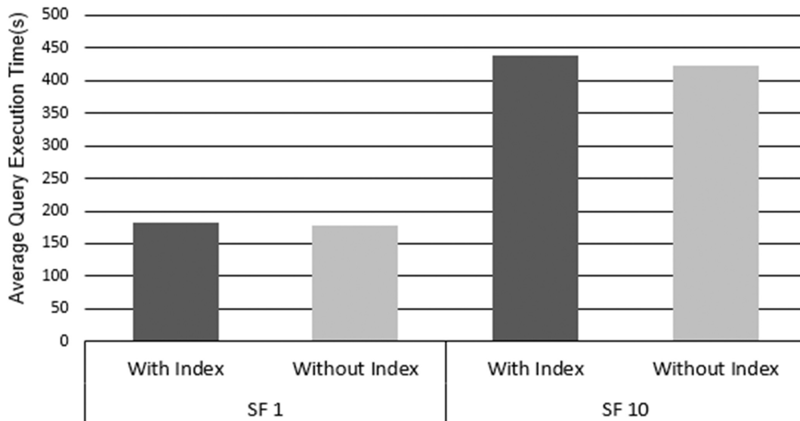


Fig. 3. Query performance with/without secondary indexes on foreign keys

Experiment 3: The effect of join algorithms
The broadcast hash join incurs out-of-memory errors for 27% of queries (6 queries) for SF 1 and 59% of queries (13 queries) for SF 10. For the other queries without any errors, the broadcast hash join improves the average query execution time by 2.0 times for both SFs 1 and 10 compared with the sort-merge join as shown in Fig. 4.

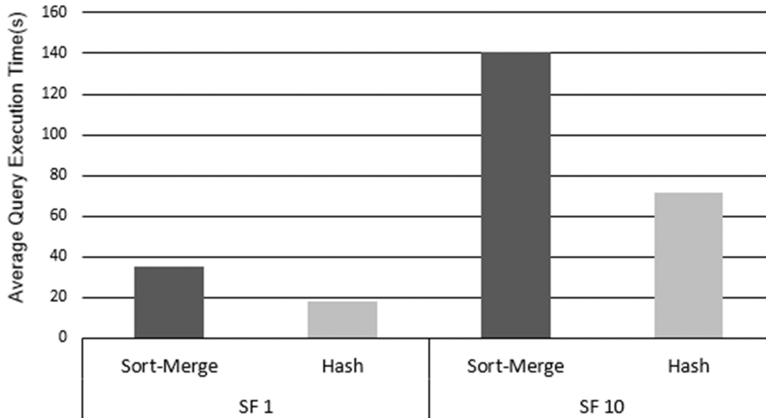


Fig. 4. Query performance with/without the USE_SORT_MERGE_JOIN hint

4.3 Discussion

Column-level denormalization with atomicity proposed for a document-oriented NoSQL is also effective for a column-oriented NoSQL. Because the secondary index is implemented outside HBase, it is not as efficient as in RDBMSs. We should use covered indexes for performance. Because the query optimizer of Phoenix does not consider the case where the broadcast hash join incurs out-of-memory errors, we often need to manually specify to use the sort-merge join. For a large database, many queries are failed due to out-of-memory errors or HRegionServer failures.

5 Conclusions

We summarized the challenges faced, lessons learned, and open problems for the migration from RDBMS to HBase with Phoenix. We addressed important issues of query translation, denormalization, secondary indexes, and join processing. Extensive experiments show that column-level denormalization with atomicity improves query performance by up to 2.7 times, the use of secondary indexes on foreign keys is not as effective as in RDBMSs, and the query optimizer of Phoenix is not very sophisticated. Important open problems for future work are supporting complex SQL queries, automatic index selection, and optimizing SQL queries for NoSQL.

References

1. Yoo, J., Lee, K.-H., Jeon, Y.-H.: Migration from RDBMS to NoSQL using column-level denormalization and atomic aggregates. J. Inf. Sci. Eng. **34**(1) (2018 to appear). http://journal.iis.sinica.edu.tw/paper/1/160464-2.pdf?cd=EF95C1D50DCDC958E
2. Karnitis, G., Arnicans, G.: Migration of relational database to document-oriented database: structure denormalization and data transformation. In: CICSyN, pp. 113–118 (2015)

3. Zhao, G., Lin, Q., Li, L., Li, Z.: Schema conversion model of SQL database to NoSQL. In: 3PGCIC, pp. 355–362 (2014)
4. Lee, C.-H., Zheng, Y.-L.: Automatic SQL-to-NoSQL schema transformation over the MySQL and HBase databases. In: IEEE ICCE-TW, pp. 426–427 (2015)
5. Zhao, G., Li, L., Li, Z., Lin, Q.: Multiple nested schema of HBase for migration from SQL. In: 3PGCIC, pp. 338–343 (2014)
6. Lee, C.-H., Zheng, Y.-L.: SQL-to-NoSQL schema denormalization and migration: a study on content management systems. In: IEEE SMC, pp. 2022–2026 (2015)
7. Vajk, T., Feher, P., Fekete, K., Charaf, H.: Denormalizing data into schema-free databases. In: IEEE CogInfoCom, pp. 747–752 (2013)
8. Vajk, T., Deak, L., Fekete, K., Mezei, G.: Automatic NoSQL schema development: a case study. In: PDCN, pp. 656–663 (2013)
9. Ho, L.-Y., Hsieh, M.-J., Wu, J.-J., Liu, P.: Data partition optimization for column-family NoSQL databases. In: IEEE Smart City, pp. 668–675 (2015)
10. Ge, W., Huang, Y., Zhao, D., Luo, S., Yuan, C., Zhou, W., Tang, Y., Zhou, J.: A secondary index with hotscore caching policy on key-value data store. In: ADMA 2014. LNCS, vol. 8933, pp. 602–615 (2014)
11. Lee, K.-H., Park, Y.-H.: Revisiting source-level XQuery normalization. IEICE Trans. Inf. Syst. **E94-D**(3), 622–631 (2011)
12. Lee, K.-H., Kim, S.-Y., Whang, E., Lee, J.-G.: A practitioner's approach to normalizing XQuery expressions. In: DASFAA 2006. LNCS, vol. 3882, pp. 437–453 (2006)
13. Kim, W.: On optimizing an SQL-like nested query. ACM Trans. Database Syst. **7**(3), 443–469 (1982)
14. Ganski, R., Wong, H.: Optimization of nested SQL queries revisited. In: ACM SIGMOD, pp. 23–33 (1987)