J Supercomput (2020) 76:7936–7950
https://doi.org/10.1007/s11227-018-2361-2

Techniques and guidelines for effective migration from RDBMS to NoSQL

Ho-Jun Kim[1] · Eun-Jeong Ko[1] · Young-Ho Jeon[1] · Ki-Hoon Lee[1]

Published online: 11 April 2018

Abstract Migration from RDBMS to NoSQL has become an important topic in a big data era. This paper provides comprehensive techniques and guidelines for effective migration from RDBMS to NoSQL. We discuss the challenges faced in translating SQL queries; the effects of denormalization, column families, secondary indexes, join algorithms, and column name length; and decision support for the migration. 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 and grouping columns into column families significantly improve query performance; the use of secondary indexes on foreign keys is not as effective as in RDBMSs; the query optimizer of Phoenix is not very sophisticated; shortened column names significantly reduce the database size and improve query performance; and the SVM classifier can predict whether query performance is improved by migration or not. Important open problems in NoSQL research are supporting complex SQL queries, automatic index selection, and optimizing SQL queries for NoSQL.

Keywords Migration · RDBMS · NoSQL · Denormalization · Column family · Secondary index · Query optimization · Decision support

This paper is an extended version of [1] which was presented at EDB 2017.

✉ Ki-Hoon Lee
kihoonlee@kw.ac.kr

1 School of Computer and Information Engineering, Kwangwoon University, 20 Kwangwoon-ro, Nowon-gu, Seoul 01897, Republic of Korea

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, 2]. 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.

In this paper, we provide comprehensive techniques and guidelines for effective 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 effects of denormalization, column families, secondary indexes, join algorithms, and column name length on NoSQL databases. Third, we propose a decision support system for the migration. 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 and grouping columns into column families significantly improve query performance; the use of secondary indexes on foreign keys is not as effective as in RDBMSs; the query optimizer of Phoenix is not very sophisticated; shortened column names significantly reduce the database size and improve query performance; and the SVM classifier can predict whether query performance is improved by migration or not. Important open problems in NoSQL research are supporting complex SQL queries, automatic index selection, and optimizing SQL queries for NoSQL.

The remainder of this paper is organized as follows. Sections 2 and 3 present background and related work, respectively. Section 4 discusses techniques and guidelines for effective migration from RDBMS to column-oriented NoSQL. Section 5 presents experimental results, and Sect. 6 provides conclusions.

2 Background

HBase is a column-oriented NoSQL and uses Hadoop Distributed File System (HDFS) as underlying storage for providing data replication and fault tolerance. In HBase, rows are ordered by the row key, and the primary key of a relational table is considered as the row key of a HBase table. Because each row can have a different set of columns, column names are repeatedly stored for each row.

HBase has a master–slave architecture as shown in Fig. 1. Rows are grouped into regions, and the columns of a region are grouped into column families. The master distributes regions to the slaves (or *region servers*) using range partitioning.

Figure 2 shows an overview of HBase I/O in a region server. HBase stores the data of a column family of a region together in a file called HFile. HFile consists of data blocks, which is a single unit of I/O. BlockCache is an LRU cache for reads, and MemStore is a write buffer. HBase reads a block from BlockCache or MemStore if it is in them, and if not, HBase reads a block from disk and caches it in BlockCache. HBase writes data to Write Ahead Log (WAL) and MemStore. The data in MemStore

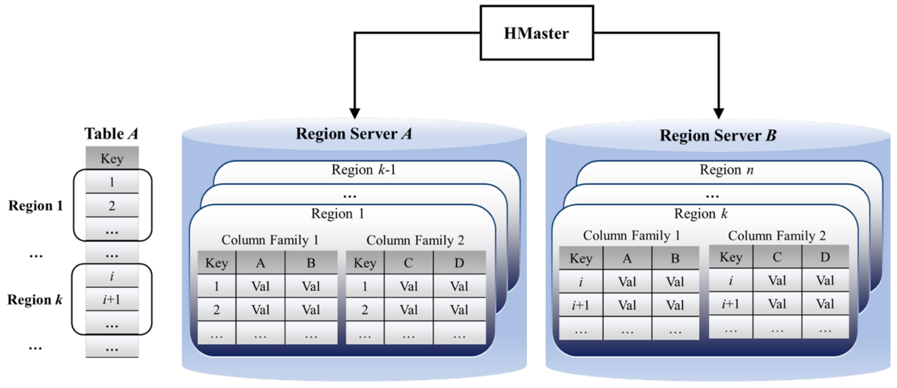

Fig. 1 HBase architecture

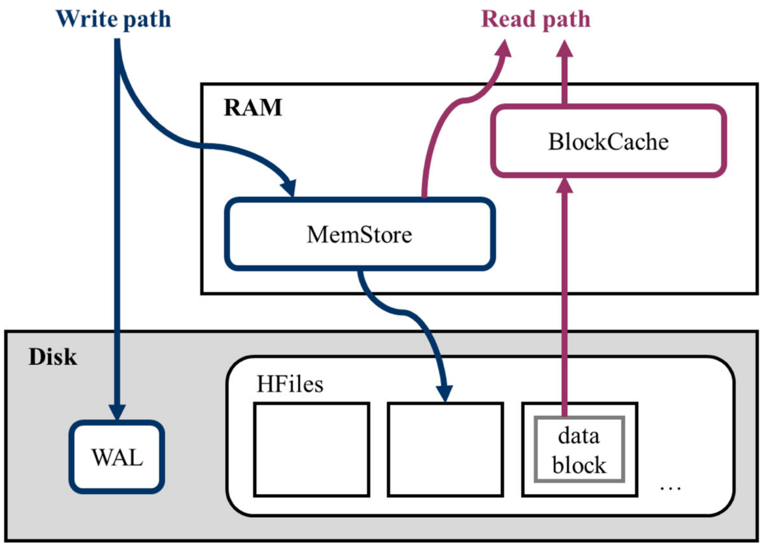

Fig. 2 HBase I/O in a region server

are sorted by the row key and flushed into HFile according to a threshold such as size limit.

HBase supports automatic sharding and failover, but it 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.

3 Related work

Reference [2] 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 [3–9], which duplicate whole tables. For column-oriented NoSQL databases, [10] proposed a column partitioning algorithm, and [11] developed a cost-based solution for schema design. Reference [12] studied the implementation of secondary indexes for HBase. Reference [13] proposed a theoretical model for implementing joins in HBase. The proposed parallel hash-join algorithm and MapReduce methods reduced the join processing time. Reference [14] proposed a three-dimensional data model in HBase for large time-series dataset analysis. The three-dimensional data model exploits the HBase version dimension in a variety of ways to provide a new perspective on data organization and management. Reference [15] performed a comparative experiment of geospatial databases and showed that Azure SQL Database is slower but more scalable than Azure DocumentDB. Reference [16] proposed MapReduce-based algorithms that parallelize data cube computation and reduce data scans. In this paper, we provide comprehensive techniques and guidelines for effective migration from RDBMS to NoSQL.

4 Migration from RDBMS to column-oriented NoSQL

In this section, we provide techniques and guidelines for effective migration from RDBMS to HBase with Phoenix. The techniques and guidelines are exemplified and discussed using a case study on TPC-H.

4.1 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 [17–20] and temporary tables. Furthermore, Phoenix simply joins tables in the order of their appearance in the FROM clause without any join reordering. To optimize a join query, we should manually rewrite the query with considering join orders.

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 process 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. All the translated TPC-H queries are posted in [21].

4.2 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*) [2], 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 such as HBase. 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. To do that, a transaction-query graph [2] is built for a given workload. A transaction-query graph contains information on primary-foreign-key relationships and transactions.

For example, Fig. 3 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`". Figure 4 shows a transaction-query graph for TPC-H Q8 where an edge denotes a primary-foreign-key relationship between two tables. A dashed edge denotes that two tables are modified within the same transaction. 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 specification, 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 `lineorders`. Thus, we can avoid "`orders` ⋈ `lineitem`" with atomicity.

4.3 Secondary indexes

Phoenix offers a secondary index on top of HBase using an index table, which consists of indexed 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.

4.4 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 region servers; each region server has a partition of the table at the left-hand side and computes the join locally. When both

```
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. 3 TPC-H Q8 [2]

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. Because query performance is significantly affected by join algorithms, it is important to select the optimal one, taking into consideration the size of memory available on the server and the sizes of both tables to join. We can force the optimizer to use a sort-merge join by using the USE_SORT_MERGE_JOIN hint.

4.5 Decision support for the migration

Because it is hard to know in advance whether query performance is improved by migration or not, we propose a decision support system for the migration. The decision support system solves a two-class classification problem: one class of queries to be improved by the migration and the other class of those not to be improved. We first build a classifier by collecting a set of training data through migration experiments. We then use the classifier to predict the class for a test query. If many queries are predicted to be improved, we can decide to migrate.

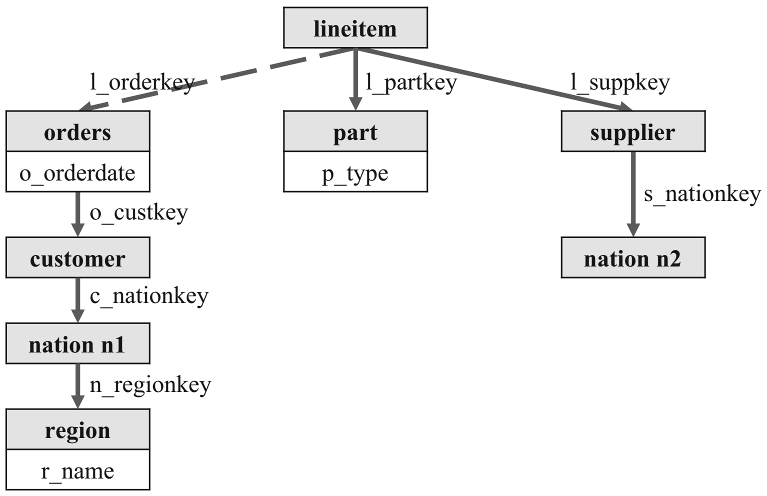

Fig. 4 Transaction-query graph for TPC-H Q8 [2]

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_regionkey_r_name
orders	o_custkey_c_nationkey o_custkey_c_mktsegment o_custkey_c_nationkey_n_name o_custkey_c_nationkey_n_regionkey_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

5 Experimental evaluation

5.1 Experimental setup

For the migration from RDBMS to HBase with Phoenix, we evaluate the effects of denormalization, column families, secondary indexes, join algorithms, and column name length on NoSQL database. Using the TPC-H benchmark with scale factor (SF) 10, we measure the query execution time for the TPC-H queries. We do not manually optimize join queries. We exclude queries that are failed due to out-of-memory errors or running more than 1000 s. 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, 8 GB of memory, Samsung 850 PRO 256 GB SSDs, and Ubuntu 16.04. For HBase, one PC is a master, and the other three PCs are region servers; we set the Java heap size, HBASE_HEAPSIZE, to 6.5 GB, the BlockCache size to 70% of HBASE_HEAPSIZE, and the MemStore size to 10% of HBASE_HEAPSIZE. We enable short-circuit reads to read data directly from disk when the data are local. For MySQL, we use only one PC; the cache size is the same as the BlockCache size, and `innodb_flush_method` is set to O_DIRECT.

We conduct the following experiments. Before starting each experiment, we reboot all the PCs and run major compaction. We use shortened column names except Experiment 5 as they are repeatedly stored.

5.1.1 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 that for the normalized schema, which has a one-to-one correspondence with the relational schema. We calculate the improvement ratio for each query that is the ratio of the execution time of the query for the normalized schema divided by that for the denormalized schema. We also compare the database size and loading time. We use the USE_SORT_MERGE_JOIN hint for all the queries and do not use secondary indexes and column families.

5.1.2 Experiment 2: the effect of column families

To see the effect of column families, we compare query performance for the denormalized schema with and without column families. The improvement ratio is calculated as the ratio of the execution time without column families divided by that with column families. We use the USE_SORT_MERGE_JOIN hint and do not use secondary indexes.

5.1.3 Experiment 3: 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. The improvement ratio is calculated as the ratio of the execution time without secondary indexes divided by that with secondary indexes. We use the denormalized schema with column families and the USE_SORT_MERGE_JOIN hint. We also run the same test for MySQL to see the effect of secondary indexes on RDBMS.

5.1.4 Experiment 4: 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. The improvement ratio is calculated as the ratio of the execution time with the USE_SORT_MERGE_JOIN hint divided by that without the USE_SORT_MERGE_JOIN hint. For comparison, we exclude queries that do not perform the broadcast hash join. We use the normalized schema without secondary indexes.

5.1.5 Experiment 5: the effect of column name length

To see the effect of column name length, we compare query performance with shortened column names (4–7 characters) and that with original column names (5–15 characters) of the TPC-H specification. The improvement ratio is calculated as the ratio of the execution time for the original column names divided by that for the shortened column names. We also compare the database size and loading time. We use the denormalized schema with column families and the USE_SORT_MERGE_JOIN hint. We do not use secondary indexes.

5.1.6 Experiment 6: decision support for migration

To collect a set of training data, we compare query performance of MySQL and HBase with Phoenix. The improvement ratio is calculated as the ratio of the execution time for MySQL divided by that for HBase with Phoenix. We classify queries according to the improvement ratio using the linear support vector machine (SVM). We generate 1100 queries (50 queries per query template) using the *qgen* query generation program of the TPC-H benchmark. Among the queries, 80% (20%) are used for training (testing). We run each query and calculate the difference between MySQL status values before and after the query is run. The calculated values are used as features. We exclude irrelevant status variables such as *Uptime*. For MySQL, we use the normalized schema with secondary indexes. For HBase with Phoenix, we use the denormalized schema with column families and the USE_SORT_MERGE_JOIN hint and do not use secondary indexes.

5.2 Experimental results

5.2.1 Experiment 1: the effect of denormalization

Figure 5 shows that the CLDA method significantly improves query performance at the expense of more space and more loading time compared with the normalization method that uses the relational schema as it is. This is because the CLDA method reduces the number of joins by duplicating columns. The maximum and average improvement ratios are 5.9 and 1.9, respectively, but the CLDA method requires 2.4 times more space and 1.8 times more loading time. The performance of queries Q1, Q6, Q14, Q15, and Q19 is degraded because the queries for the normalized schema

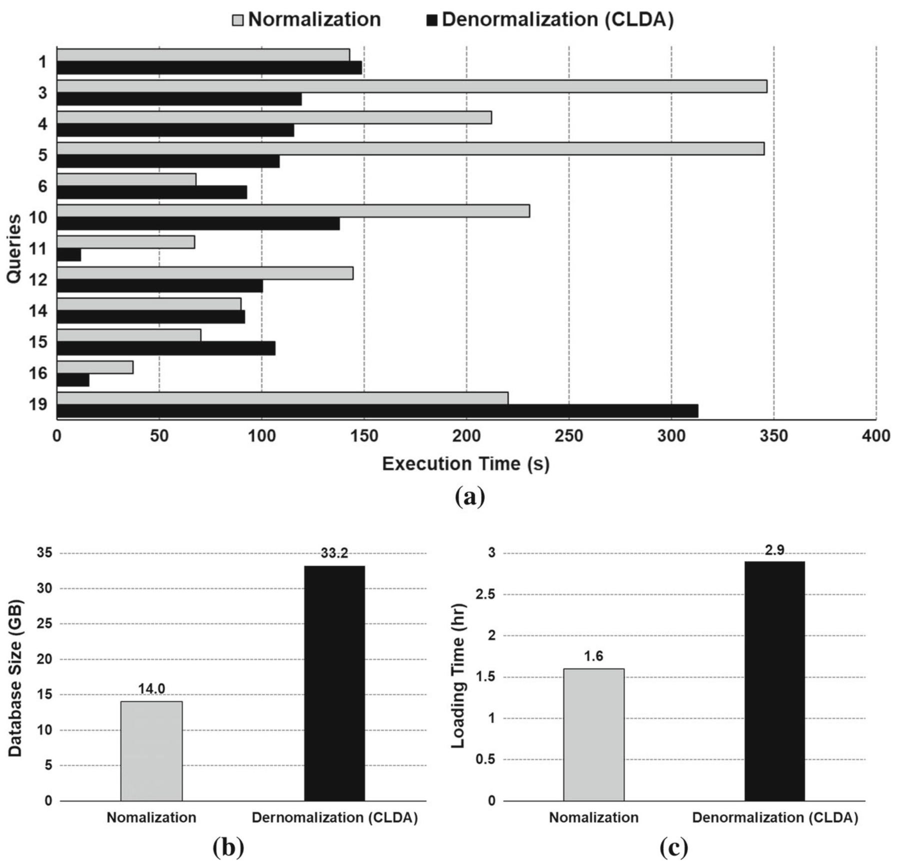

Fig. 5 Effect of column-level denormalization with atomicity. **a** Query performance with/without CLDA. **b** Database size with/without CLDA. **c** Database loading time with/without CLDA

have zero or few joins and those for the denormalized schema access one big pre-joined table `lineorders`. Because the `lineitem` and `orders` tables are combined into the `lineorders` table, the queries for the denormalized schema cannot access the `lineitem` table separately, but the queries for the normalized schema can. Thus, the CLDA method incurs more I/Os. Queries Q2, Q7, Q8, Q9, Q17, Q20, Q21, and Q22 failed for the normalization method; and queries Q13, Q17, Q18, Q20, Q21, and Q22 failed for the CLDA method. The common cause of the failures is that the intermediate results of joins consume a lot of memory and incur out-of-memory errors. We exclude the failed queries when calculating the improvement ratio.

5.2.2 Experiment 2: the effect of column families

Figure 6 shows that denormalization with column families significantly improves query performance compared with that without column families. Because columns with similar access patterns are grouped into a column family and stored together on

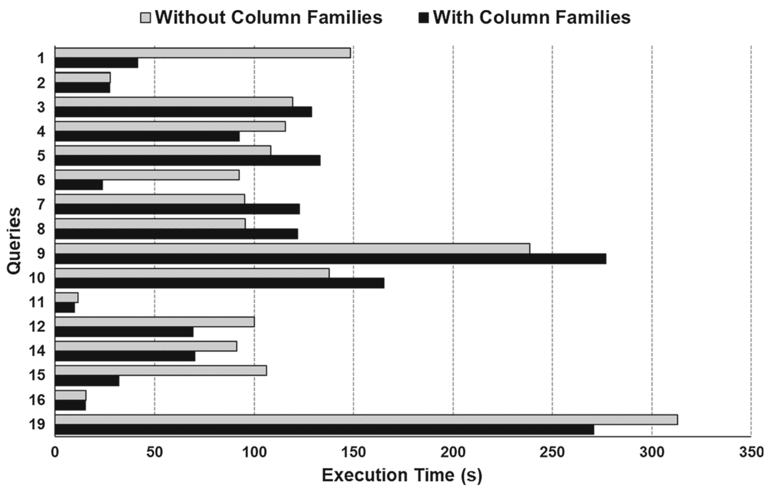

Fig. 6 Query performance with/without column families

disk, the number of disk accesses is reduced. The maximum and average improvement ratios are 3.9 and 1.5, respectively. The database size with column families is almost the same with that without column families, and the database loading time is 1.24 times longer for with column families. The performance of queries Q3, Q5, Q7, Q8, Q9, and Q10 is degraded because the queries access more than one column family and incur more I/Os compared with the counterparts without column families. Queries Q13, Q17, Q18, Q20, Q21, and Q22 failed.

5.2.3 Experiment 3: 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 1 h, and the average query execution time of the other 27% (six 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 7 shows that for HBase with Phoenix, the query execution times with and without secondary indexes are almost the same. The maximum and average improvement ratios are 1.12 and 0.96, respectively. The reason is that secondary indexes are not used for most of the queries because they are not estimated to be beneficial. For some queries, secondary indexes degrade performance even if the queries do not use the indexes because the indexes consume space in the cache. Queries Q13, Q17, Q18, Q20, Q21, and Q22 failed.

5.2.4 Experiment 4: the effect of join algorithms

For the normalized schema without secondary indexes, the sort-merge join incurs out-of-memory errors for 36% of queries (eight queries), but the broadcast hash join 64% of queries (14 queries). Figure 8 shows that for the six queries that successfully perform the broadcast hash join, the maximum and average improvement ratios are

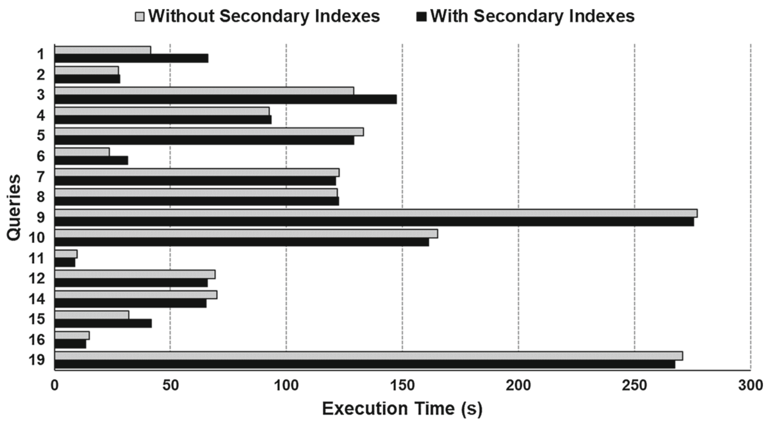

Fig. 7 Query performance with/without secondary indexes on foreign keys

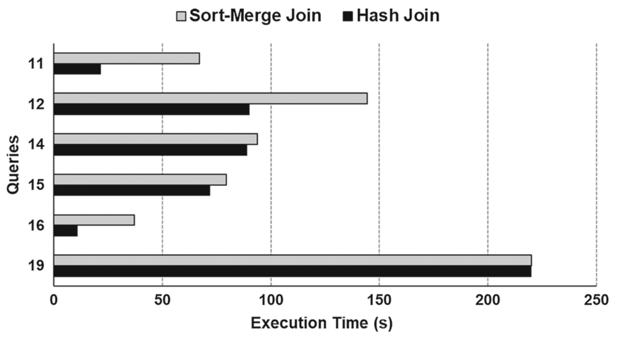

Fig. 8 Query performance with/without the USE_SORT_MERGE_JOIN hint

3.4 and 1.9, respectively. The broadcast hash join is faster than the sort-merge join because the broadcast hash join is an in-memory operation.

5.2.5 Experiment 5: the effect of column name length

Figure 9 shows that compared with original column names, shortened column names significantly reduce the database size and improve query performance. The maximum and average improvement ratios are 2.3 and 1.2, respectively. The loading time is almost the same because of parallel writes in a region server, but original column names show higher I/O utilization compared with shortened column names. Queries Q13, Q17, Q18, Q20, Q21, and Q22 failed.

5.2.6 Experiment 6: decision support for the migration

Figure 10 shows that compared with the MySQL, HBase with Phoenix improves the performance of Q1, Q3, Q6, Q9, and Q15, but degrades the performance of Q2, Q4, Q5, Q7, Q8, Q10, Q11, Q12, Q14, Q16, and Q19 and fails to execute Q13, Q17,

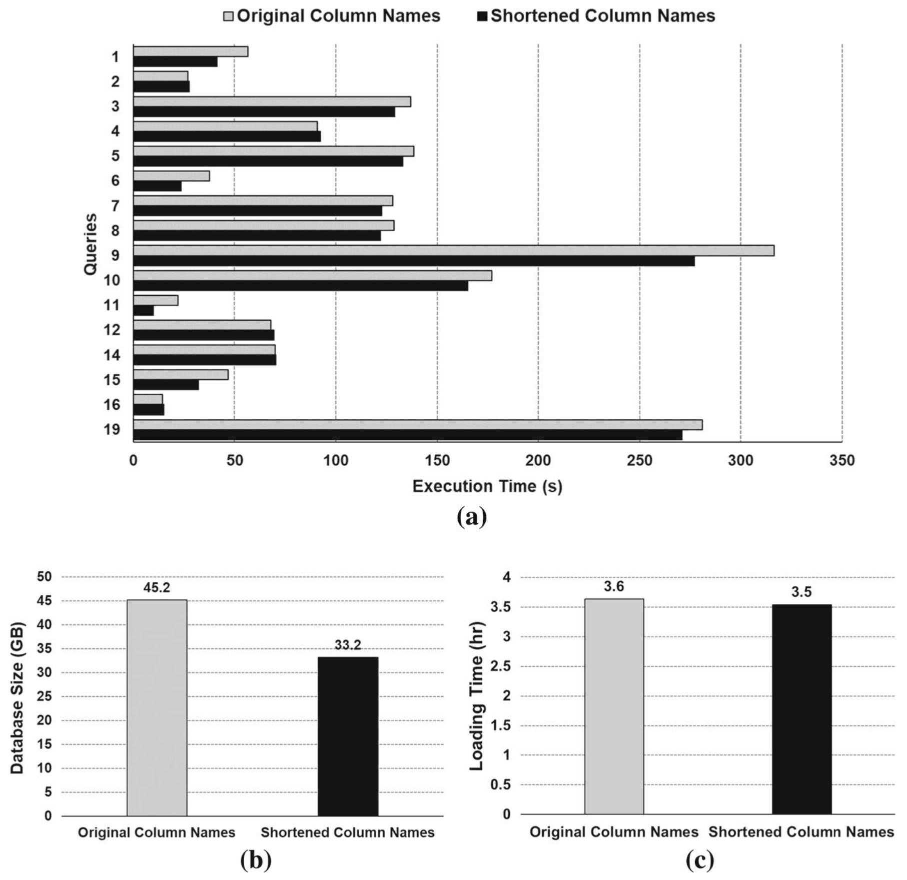

Fig. 9 Effect of column name length. **a** Query performance with original/shortened column names. **b** Database size. **c** Database loading time

Q18, Q20, Q21, and Q22. The maximum and average improvement ratios are 3.4 and 0.9, respectively. The accuracy of the proposed linear SVM classifier is 100%. We can precisely predict the improvement in query performance after migration by the virtue of three features—Innodb_rows_read, Innodb_pages_read, and Open_tables. Innodb_rows_read is the number of rows read, Innodb_pages_read the number of pages read from the buffer pool, and Open_tables the number of tables opened.

5.3 Guidelines

The CLDA method proposed for a document-oriented NoSQL is also effective for a column-oriented NoSQL. Using column families significantly improves query performance. Because the secondary index is implemented outside HBase, it is not as efficient as in RDBMSs. We should use covered indexes for performance. The query optimizer of Phoenix does not consider join reordering. It simply joins tables in the

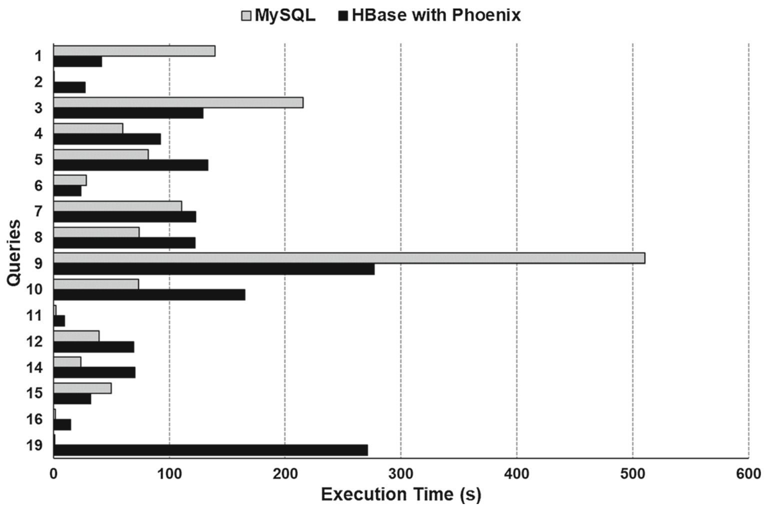

Fig. 10 Query performance of MySQL and HBase with Phoenix

order of their appearance in the FROM clause and often incurs Cartesian product (e.g., Q2, Q8, and Q9 on the normalized schema). Furthermore, 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. Shortened column names significantly reduce the database size and improve query performance. The SVM classifier precisely predicts whether query performance is improved by migration or not. HBase with Phoenix outperforms MySQL when the number of I/O requests and the number of table joins are large.

6 Conclusions

For effective migration from RDBMS to HBase with Phoenix, we provided techniques and guidelines for query translation, denormalization, column families, secondary indexes, join algorithms, column name length, and decision support for the migration. Experimental results show that denormalization, column families, and shortened column names improve query performance; shortened column names reduce the database size; and the SVM classifier can support the decision of migration. Important open problems for future work are supporting complex SQL queries, automatic index selection, and optimizing SQL queries for NoSQL.

Acknowledgements This work was supported by the National Research Foundation of Korea(NRF) Grant funded by the Korea government (MSIT) (No. NRF-2015R 1C 1A 1A02036517). The present Research has been conducted by the Research Grant of Kwangwoon University in 2017.

References

1. Kim H-J, Ko E-J, Jeon Y-H, Lee K-H (2017) Migration from RDBMS to column-oriented NoSQL: lessons learned and open problems. In: EDB, LNEE, vol 461, pp 25–33

2. Yoo J, Lee K-H, Jeon Y-H (2018) Migration from RDBMS to NoSQL using column-level denormalization and atomic aggregates. J Inf Sci Eng 34(1):243–259
3. Karnitis G, Arnicans G (2015) Migration of relational database to document-oriented database: structure denormalization and data transformation. In: CICSyN, pp 113–118
4. Zhao G, Lin Q, Li L, Li Z (2014) Schema conversion model of SQL database to NoSQL. In: 3PGCIC, pp 355–362
5. Lee C-H, Zheng Y-L (2015) Automatic SQL-to-NoSQL schema transformation over the MySQL and HBase databases. In: IEEE ICCE-TW, pp 426–427
6. Zhao G, Li L, Li Z, Lin Q (2014) Multiple nested schema of HBase for migration from SQL. In: 3PGCIC, pp 338–343
7. Lee C-H, Zheng Y-L (2015) SQL-to-NoSQL schema denormalization and migration: a study on content management systems. In: IEEE SMC, pp 2022–2026
8. Vajk T, Feher P, Fekete K, Charaf H (2013) Denormalizing data into schema-free databases. In: IEEE CogInfoCom, pp 747–752
9. Vajk T, Deak L, Fekete K, Mezei G (2013) Automatic NoSQL schema development: a case study. In: PDCN, pp 656–663
10. Ho L-Y, Hsieh M-J, Wu J-J, Liu P (2015) Data partition optimization for column-family NoSQL databases. In: IEEE Smart City, pp 668–675
11. Mior MJ, Salem K, Aboulnaga A, Liu R (2016) NoSE: schema design for NoSQL applications. In: IEEE ICDE, pp 181–192
12. Ge W, Huang Y, Zhao D, Luo S, Yuan C, Zhou W, Tang Y, Zhou J (2014) A secondary index with hotscore caching policy on key-value data store. In: ADMA, LNCS, vol 8933, pp 602–615
13. Gadkari A, Nikam VB, Meshram BB (2014) Implementing joins over HBase on cloud platform. In: IEEE CIT, pp 547–554
14. Han D, Stroulia E (2012) A three-dimensional data model in HBase for large time-series dataset analysis. In: IEEE MESOCA, pp 47–56
15. Baralis E, Valle AD, Garza P, Rossi C, Scullino F (2017) SQL versus NoSQL databases for geospatial applications. In: IEEE BSD
16. Lee S-A, Kim J-H, Moon Y-S, Lee W-K (2015) Efficient level-based top-down data cube computation using MapReduce. In: Transactions on Large-Scale Data- and Knowledge-Centered Systems XXI, LNCS, vol 9260, pp 1–19
17. Lee K-H, Park Y-H (2011) Revisiting source-level XQuery normalization. IEICE Trans Inf Syst E94-D(3):622–631
18. Lee K-H, Kim S-Y, Whang E, Lee J-G (2006) A practitioner's approach to normalizing XQuery expressions. In: DASFAA, LNCS, vol 3882, pp 437–453
19. Kim W (1982) On optimizing an SQL-like nested query. ACM Trans database Syst 7(3):443–469
20. Ganski R, Wong H (1987) Optimization of nested SQL queries revisited. In: ACM SIGMOD, pp 23–33
21. TPC-H Queries, https://sites.google.com/site/kwunivdsl