JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 34, 243-259 (2018)
DOI: 10.6688/JISE.2018.34.1.15

Migration from RDBMS to NoSQL Using Column-Level Denormalization and Atomic Aggregates*

JAEJUN YOO[1], KI-HOON LEE[2,+] AND YOUNG-HO JEON[2]
[1]*Positioning/Navigation Technology Research Section*
Intelligent Cognitive Technology Research Division
Electronics and Telecommunications Research Institute
Daejeon, 34129 Korea
E-mail: jjryu@etri.re.kr
[2]*Department of Computer Engineering*
Kwangwoon University
Seoul, 01897 Korea
E-mail: {kihoonlee, pointnb}@kw.ac.kr

As NoSQL grows in popularity, many organizations are attempting to migrate their databases from RDBMS to NoSQL. Because NoSQL is very different from RDBMS, such migration is a challenging issue. For example, NoSQL does not support join operations or transactions. We propose a novel solution for database migration from RDBMS to document-oriented NoSQL, which is the most widely used type of NoSQL. Our method not only avoids join operations with a marginal increase in the database size, but also supports atomicity using the notions of column-level denormalization and atomic aggregates. *Column-level denormalization* duplicates only columns that are accessed in non-primary-foreign-key-join predicates. *Atomic aggregates* combine tables that are modified within the same transaction into a unit of atomic updates called an aggregate. Experimental results using TPC-H and MongoDB show that our method improves the query performance by up to 2.2 times using 1.5 times more space compared with a baseline method that uses the relational schema as it is. Compared with the state-of-the-art method, which duplicates whole tables to avoid join operations, our method improves the query performance by up to 2.0 times with 2.8 times less space.

***Keywords*:** database migration, column-level denormalization, atomicity, RDBMS, NoSQL

1. INTRODUCTION

NoSQL systems such as MongoDB, Cassandra, HBase, and Redis offer an alternative to the classic RDBMS, and are growing in popularity. NoSQL systems sacrifice consistency for other factors such as availability, scalability, and performance, which are more important to big data applications [1]. To manage big data challenges, many organizations want to migrate their databases from RDBMS to NoSQL, especially for workloads such as OLAP that do not involve many update transactions [2]. Currently, much of the database migration is conducted manually because the migration process is very complicated to fully automate owing to the different design principles, data models, and fea-

Received June 22, 2016; revised September 6, 2016; accepted November 25, 2016.
Communicated by Wei-Shinn Ku.
* 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 2016.
+ Corresponding author.

tures. For example, NoSQL systems do not support join operations or ACID transactions.

The simplest method for a migration would be to create a NoSQL schema that has a one-to-one correspondence with the relational schema, which is often normalized. We call this a *normalization method*. A normalization method often results in a poor query performance for NoSQL because join operations are rarely supported in NoSQL systems and should be processed at the application layer in a much more expensive way than RDBMS. Another drawback of a normalization method is that it does not provide any support for transactions.

Denormalization, which duplicates data to minimize the number of join operations, is preferred over normalization in NoSQL data modeling. Compared with normalization, denormalization often accelerates queries involving many join operations. However, denormalization often slows down updates, degrades data integrity, and slows down queries that do not involve many join operations.

We propose a novel method for database migration from RDBMS to NoSQL that significantly reduces the disadvantages of naive denormalization and supports atomicity for transactions. We call our method *Column-Level Denormalization with Atomicity* (*CLDA*). To the best of our knowledge, this is the first work conducted towards this objective. First, we propose a new notion called column-level denormalization. Column-level denormalization duplicates only those columns that are accessed in non-primary-foreign-key-join predicates to avoid join operations with a minimal increase of data redundancy. Example 1 shows that column-level denormalization can avoid join operations without denormalizing entire tables. Second, based on column-level denormalization, we propose a schema migration method that fully exploits the atomic update feature provided by NoSQL to support the implementation of transactions. Example 2 shows how we can support atomicity for the TPC-H Benchmark [3]. Third, we conducted extensive experiments using TPC-H and MongoDB [4]. The results show that our method improves the query performance by up to 2.2 times using 1.5 times more space compared with a normalization method. Compared with the state-of-the-art method [5] that denormalizes whole tables, our method improves the query performance by up to 2.0 times with a 2.8 times less space.

Example 1: For the TPC-H Q8 shown in Fig. 1, the columns `r_name`, `o_orderdate`, and `p_type` appear in non-primary-foreign-key-join predicates, which are shaded. If we add `r_name` to the `orders` table and `p_type` to the `lineitem` table, we can avoid "`orders` ⋈ `customer` ⋈ `nation` ⋈ `region`" and "`lineitem` ⋈ `part`."

Example 2: The TPC-H specification requires that the `lineitem` and `orders` tables are modified within the same transaction. To support transaction-like behavior in NoSQL applications, the schema migration algorithm should automatically combine the `lineitem` and `orders` tables into a unit of atomic updates.

In this paper, we focus on document-oriented NoSQL, particularly MongoDB, because it is the most widely used NoSQL system [6]. The database migration process is conducted in two phases: schema migration and data migration [7]. We focus on the schema migration phase because automating data migration based on the results of schema migration is rather straightforward.

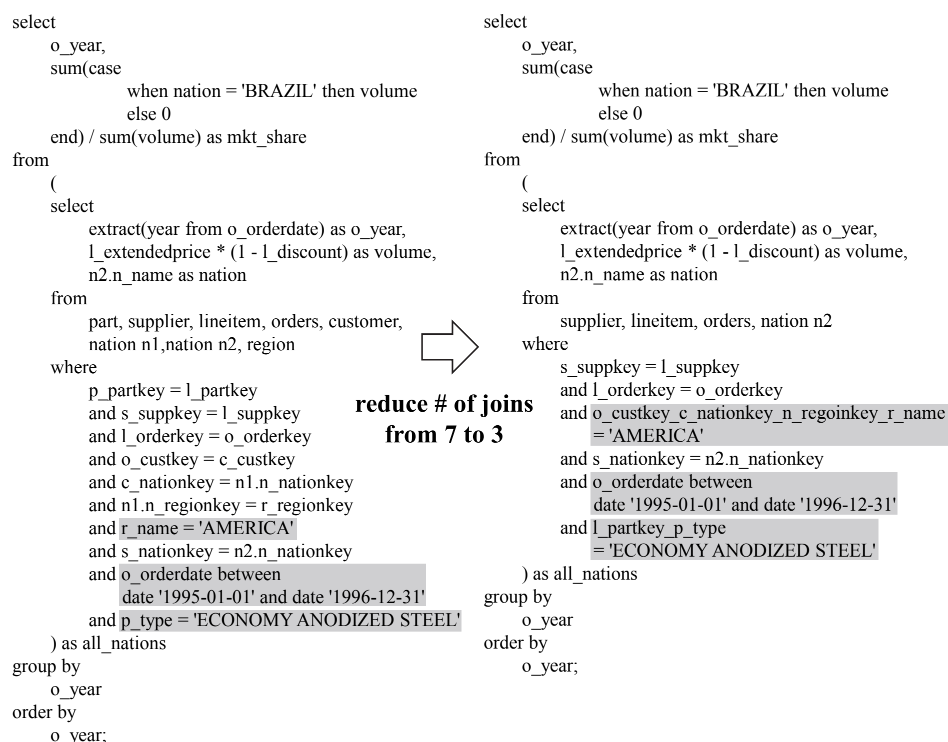

Fig. 1. TPC-H Q8 before and after column-level denormalization.

The rest of this paper is organized as follows. Section 2 introduces NoSQL data modeling and document-oriented NoSQL. Section 3 proposes a naive migration method based on table-level denormalization, and Section 4 proposes a novel migration method based on column-level denormalization and atomic aggregates. Section 5 compares the methods, and Section 6 reviews existing work. Section 7 presents our experimental results. Finally, Section 8 provides some concluding remarks.

2. BACKGROUND

2.1 NoSQL Data Modeling

NoSQL is an umbrella term for all data stores that do not follow a relational data model and generally do not support SQL for data manipulation [8]. NoSQL data modeling often starts from application-specific queries, as opposed to relational modeling, which is driven by the structure of the data itself [9].

In addition to denormalization, aggregate orientation is a common characteristic of NoSQL data models [10]. An *aggregate* is a collection of related objects, which is a unit of atomic updates [10]. In general, NoSQL systems do not support atomic transactions that span multiple aggregates [10]. If a set of aggregates should be atomically modified,

we have to combine the set of aggregates into a single aggregate, which is called an *atomic aggregate* [9, 11]. Although atomic aggregates are not a complete solution for a transaction [9, 11] they can be used as building blocks to implement transactions at the application layer.

2.2. Document-Oriented NoSQL

Document-oriented NoSQL systems store data in a document format such as JSON. A document is an aggregate and allows nested subdocuments and arrays. MongoDB is the most popular document-oriented NoSQL [6]. Table 1 compares the terminology used in MongoDB with that in RDBMS. In contrast to RDBMS, documents within a single collection can have a different set of fields [4]. In practice, documents within a collection have a similar schema [4]. The primary-foreign key relationship can be expressed as nested subdocuments or references. We can conduct join operations at the application layer using references.

Table 1. Terminology comparison between RDBMS and MongoDB.

RDBMS	MongoDB
Table	collection
Row	document
column	field
primary-foreign key relationship	nested subdocument or reference

3. TABLE-LEVEL DENORMALIZATION

A naive method for avoiding join operations would be to denormalize whole tables by joining tables along primary-foreign key relationships. If the relationship is one-to-many, tuples in the table with the primary key are duplicated in the denormalized table. We call this method *table-level denormalization*. To describe a schema migration algorithm that uses table-level denormalization, we first define a schema graph in Definition 1.

Definition 1: For a given relational schema, RS, the *schema graph* $G = (N, E)$ is defined as follows: A node $n \in N$ corresponds to a table $t \in RS$. An edge $e \in E$ corresponds to a primary-foreign key relationship between two different tables and is directed from the foreign key table to the primary key table. If a foreign key is a subset of another composite foreign key, the schema graph does not have an edge for the former foreign key.

Example 3: Fig. 2 shows the schema graph for the TPC-H schema. There is no edge from `lineitem` to `supplier` because the foreign key `l_suppkey` in `lineitem` is a subset of the composite foreign key (`l_partkey`, `l_suppkey`) that refers to `partsupp`. For the same reason, there is no edge from `lineitem` to `part`.

If the schema graph is acyclic, we transform it into a set of schema trees by duplicating nodes that have multiple incoming edges. A node with no incoming edges in the schema graph will be the root of a schema tree. If the schema graph is cyclic, we can

make it acyclic by deleting edges based on the user's decision. The development of algorithms that automatically make the schema graph acyclic is out of the scope of this paper because table-level denormalization is not a point of focus.

Example 4: Fig. 3 shows a schema tree for a TPC-H schema. The `nation` and `region` nodes are duplicated, and the `lineitem` node is the root node.

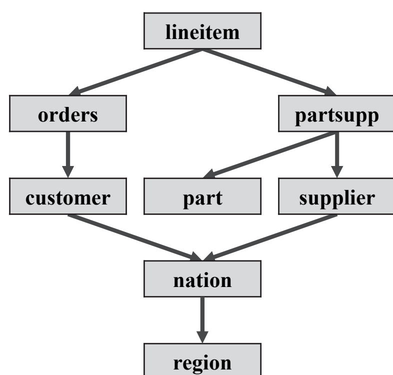

Fig. 2. Schema graph for the TPC-H schema.

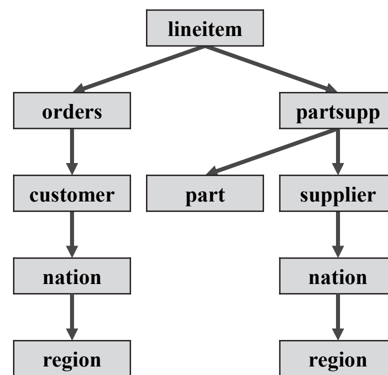

Fig. 3. Schema tree for the TPC-H schema.

Algorithm 1 shows a schema migration algorithm that uses table-level denormalization. We first generate a schema graph from the relational schema and make it acyclic if needed. We then transform the schema graph into a set of schema trees. For each schema tree, we create a collection for the root node and replace a foreign key in each node with the child node that the foreign key refers to (*i.e.*, primary key table).

Example 5: Fig. 4 shows the MongoDB schema migrated from the TPC-H schema using Algorithm 1. There is only one collection: `lineitem`.

Algorithm 1: A schema migration algorithm that uses table-level denormalization
Input: relational schema *RS*
Output: MongoDB schema
1. Generate a schema graph *G* from *RS*
2. Make *G* acyclic based on user's decision if needed
3. Transform *G* into a set *ST* of schema trees
4. **for** (each schema tree $T \in ST$) {
5. create a collection for the root of *T*
6. **for** (each non-root node *n* of *T*) {
7. embed *n* into the parent node n_p of *n*
8. remove the foreign key in n_p that refers to *n*
9. }
10. }

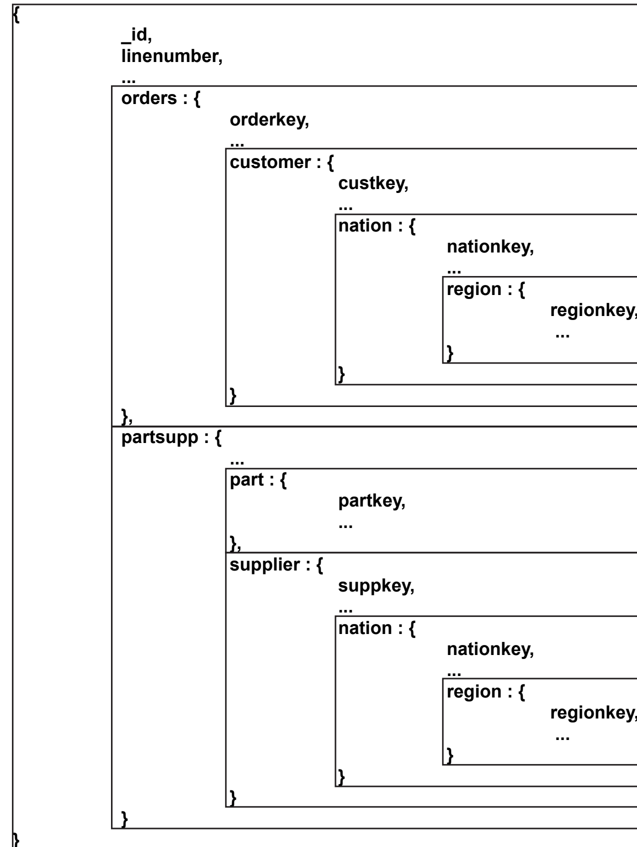

Fig. 4. MongoDB schema migrated from the TPC-H schema using Algorithm 1.

4. COLUMN-LEVEL DENORMALIZATION WITH ATOMICITY

Table-level denormalization incurs a large amount of data redundancy because it duplicates entire tables. Furthermore, it does not consider the support of atomicity and limits data accessibility. In this section, we propose a novel method, *column-level denormalization with atomicity* (*CLDA*), which overcomes the problems of the table-level denormalization method.

4.1 Column-Level Denormalization

Column-level denormalization maintains all of the original tables and duplicates only a few of the columns. The challenging issue here is the criterion for selecting the columns to be duplicated. To the best of our knowledge, such criterion has not been previously addressed in the literature.

We propose a criterion that duplicates only columns that appear in *non-primary-foreign-key-join predicates* defined in Definition 2.

Definition 2: For a given query, a *primary-foreign-key-join predicate* is a predicate with a primary key column and a foreign key column referring to the primary key column. A *non-primary-foreign-key-join predicate* is a predicate that is not a primary-foreign-key-join predicate and appears in the FROM or WHERE clauses.

Using this criterion, we can avoid most of the join operations in the predicate evaluation phase, as shown in Example 1. There are cases in which we need join operations to access columns that do not appear in non-primary-foreign-key-join predicates, for example, `n2.n_name` in Fig. 1. However, in these cases, the cost of the join operation is often very small because we can reduce the input size of the join operation by applying predicates as early as possible.

To describe the column-level denormalization algorithm, we first define the *transaction-query graph* in Definition 3.

Definition 3: For a given query, q, the *transaction-query graph* $G=(N, E)$ for q is defined as follows: A node $n \in N$ corresponds to table t mentioned in q. Tables with the same name are distinguished by their tuple variables. Node n has columns that appear in non-primary-foreign-key-join predicates on table t. An edge $e \in E$ corresponds to an inner join predicate between table t on a primary key and table u on a foreign key and is directed from u to t. Here, $t \neq u$. We label the edge with the foreign key. If tables t and u are modified within the same transaction, the edge is shown by a dashed line; otherwise, it is shown by a solid line. A transaction-query graph does not have edges for correlated join predicates in the subqueries.

Example 6: Fig. 5 shows a transaction-query graph for TPC-H Q8. The nodes `region`, `part`, and `orders` have columns showing non-primary-foreign-key-join predicates. The nodes `lineitem` and `orders` are modified within the same transaction.

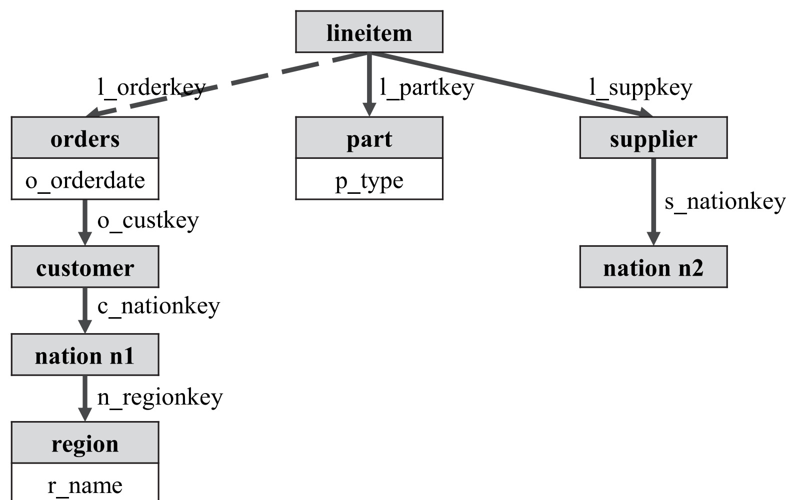

Fig. 5. Transaction-query graph for TPC-H Q8.

Algorithm 2 shows a column-level denormalization algorithm. For each query q of a given workload, we build a transaction-query graph $G(N, E)$. For each $n \in N$ with no incoming solid-line edge, we find a set S of reachable nodes through solid-line edges and add the columns of the nodes in S to table n. We explain how we handle dashed lines in the next section. Because there are common columns appearing in several non-primary-foreign-key-join predicates, the number of duplicated columns is often small.

Algorithm 2: A column-level denormalization algorithm
Input: a set Q of queries of a given workload
Output: denormalized relational schema
1. **for** (each query $q \in Q$) {
2. build a transaction-query graph $G(N, E)$ for q
3. **for** (each node $n \in N$ that has no incoming solid-line edge) {
4. collect a set S of reachable nodes via solid-line edges
5. **for** (each node $s \in S$) {
6. add the columns of the node s to the table n
7. }
8. }
9. }

Example 7: Using the transaction-query graph in Fig. 5, Algorithm 2 adds `l_partkey_p_type` to the `lineitem` table and `o_custkey_c_nationkey_n_regionkey_r_name` to the `orders` table. Here, the name of each added column is prefixed by the names of the foreign keys that appear on the path to n. Note that the edge between `lineitem` and `orders` is not a solid line. After the column-level denormalization, we can avoid "`lineitem ⋈ part`" and "`orders ⋈ customer ⋈ nation ⋈ region.`" We still need to handle "`orders ⋈ lineitem ⋈ supplier ⋈ nation.`" We can avoid "`orders ⋈ lineitem`" using atomic aggregates explained in the next section. The cost of "`<intermediate result> ⋈ supplier ⋈ nation`" is small because we can reduce the size of the intermediate result by applying predicates early.

4.2 Schema Migration Algorithm Based on Atomic Aggregates

Algorithm 3 shows a schema migration algorithm that uses atomic aggregates to support transaction-like behavior in NoSQL applications. When two tables are modified within the same transaction, we need to combine them into a single collection to ensure atomicity. Suppose that the two tables have a primary-foreign key relationship. When a user attempts to delete row r_t in table t with a primary key referenced by a foreign key in another table u, we need to atomically delete not only r_t but also all rows in u that reference r_t. A similar process is applied for any updates. We can ensure the atomicity by creating a collection c_t for t and nesting u into c_t as an array of subdocuments. When two tables are modified within the same transaction but do not have a primary-foreign key relationship, we do not combine the tables because it is hard to automatically determine how to combine such tables. In this case, the user should conduct transaction support at

the application layer. Finally, we create a collection for each table that does not have other tables modified within the same transaction.

Algorithm 3: A schema migration algorithm that uses atomic aggregates
Input: a relational schema *RS* denormalized by Algorithm 2, a workload on *RS*
Output: MongoDB schema

```
 1. for (each table t∈RS) {
 2.     if  (∃ u∈RS such that t and u are modified within the same transaction) {
 3.         if  (the foreign key in u refers to the primary key in t) {
 4.             create a collection c_t for t
 5.             nest u into c_t as an array of subdocuments
 6.             remove the foreign key in u
 7.         }
 8.         else if (the foreign key in t refers to the primary key in u) {
 9.             create a collection c_u for u
10.             nest t into c_u as an array of subdocuments
11.             remove the foreign key in t
12.         }
13.         else {
14.             create a collection c_t for t
15.             create a collection c_u for u
16.         }
17.          RS = RS – {u}
18.     }
19.     else {
20.         create a collection c_t for t
21.     }
22. }
```

Example 8: For the TPC-H benchmark, Algorithm 3 combines the `lineitem` and `orders` tables into a single collection, `orders`, as shown in Fig. 6. It also creates a collection for each of the other tables: `partsupp`, `part`, `supplier`, `customer`, `nation`, and `region`.

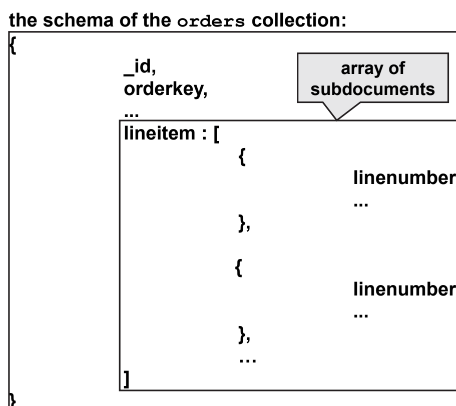

Fig. 6. Result of Algorithm 3 for the TPC-H benchmark.

5. COMPARISON OF THE METHODS

Table 2 compares the three schema migration methods. The comparison of the number of join operations is obvious. We note that even the table-level denormalization method cannot remove correlated join operations involving subqueries. For a one-to-many relationship, the table-level denormalization method duplicates tuples in the one side. When a query needs to access the one side separately, we need an *extract* operation, which projects a set of fields belonging to the one side and eliminates duplicated tuples. For example, when we want to access only `customer` in Fig. 4, we should extract it from `lineitem` by eliminating duplicated customers. Duplicate elimination often requires expensive sorting operations. While extract operations are required for the table-level denormalization method, *unwind* operations [4] are required for our method because our method creates an array of subdocuments such as `lineitem` in Fig. 6 to support atomicity. An unwind operation is less expensive than an extract operation because an unwind operation can be implemented with a single scan. The number of the additional operations (extract or unwind) of each denormalization method depends on queries. The normalization method does not incur any extract or unwind operations, but incurs many join operations.

Table 2. Comparison of the schema migration methods.

Criterion \ Method	Normalization	Column-level Denormalization with Atomicity	Table-level Denormalization
the number of join operations	large	medium	small
extract operation	not required	not required	required
unwind operation	not required	required	not required
data accessibility	high	medium	low
support for atomicity	no	yes	no
update cost	low	medium	high
the numbers of memory and disk I/Os	small	medium	large
database and document size	small	medium	large
data access and processing cost	low	medium	high

Denormalization restricts data accessibility, and lack of data accessibility forces inefficient queries. For example, a query performs join operations could be more efficient than a query scans a big pre-joined collection, but the former is often not a possible alternative for the denormalization methods. The data accessibility of the table-level denormalization method is lowest because all queries should access one big pre-joined collection. Our method has a higher data accessibility than the table-level denormalization method because it keeps most of the normalized tables and thus allows performing join operations for those tables if needed.

The data redundancy of the table-level denormalization method is the highest. Redundant data not only increase the update cost, but also waste CPU cache and

main memory cache and incur more cache misses, resulting in more memory and disk I/Os. Furthermore, higher data redundancy results in larger database and document size, and higher data access and processing cost.

Compared with normalization method, the advantage of the denormalization methods is derived from the smaller number of join operations. However, the denormalization methods have disadvantages such as additional operations, lower data accessibility, higher update cost, more memory and disk I/Os, larger database and document size, and higher data access and processing cost.

Compared with the table-level denormalization method, our method not only support atomicity but also has higher data accessibility, lower update cost, less memory and disk I/Os, smaller database and document size, and lower data access and processing cost. The disadvantage of our method is derived from the larger number of join operations, but the increase of the number is often small. Our method has to do the denormalization process whenever a new query is added to the workload whereas the table-level denormalization method does not. Thus, our method is well suitable for database applications with relatively static workload. In practice, most of the database users (*e.g.*, reservation agents) access the database through previously implemented and tested programs, and thus, the workload is predefined and rarely changed.

6. RELATED WORKS

Although document-oriented NoSQL is one of the most widely used types of NoSQL, there have been very few studies on database migration from RDBMS to document-oriented NoSQL. Zhao *et al.* [12], and Karnitis and Arnicans [5] proposed variants of the table-level denormalization method. The most recent work [5] proposed an algorithm based on a breadth-first search (BFS) algorithm that visits each edge exactly once to do not embed too many nodes. For example, in Fig. 2, the edge from `nation` to `region` is visited only once, and thus, `region` is embedded only once.

There have been research efforts on the migration from RDBMS to column-oriented NoSQL such as HBase and Cassandra, which is most similar to the traditional RDBMS [13]. Li [14], and Schram and Anderson [15] presented case studies. Zhao *et al.* [16], Lee and Zheng [13, 17], and Vajk *et al.* [18, 19] proposed variants of the table-level denormalization method. These methods for column-oriented NoSQL are not directly applicable to document-oriented NoSQL because their data models are different. Furthermore, these methods are not column-level denormalization.

In a relational context, Shin and Sanders [20] provide comprehensive guidelines for denormalization. They summarize the general models for denormalization: collapsing relations (CR), partitioning a relation (PR), adding redundant attributes (RA), and adding derived attributes (DA). A table-level denormalization method belongs to the CR model, and a column-level denormalization method belongs to the RA model. Li and Patel [21] proposed a table-level denormalization method for an in-memory RDBMS using techniques such as columnar storage, dictionary encoding, and packed code scan. In this paper, we provide a column-level denormalization algorithm for database migration from RDBMS to document-oriented NoSQL.

Several efforts have been made to evaluate the performance of NoSQL systems. Floratou *et al.* [22] compared the performance of NoSQL systems (Hive and MongoDB)

and RDBMSs (SQL Server PDW and SQL Server) and found that the RDBMSs still have significant advantages, but the NoSQL systems are competitive in some cases. Klein *et al.* [23] evaluated three NoSQL systems (MongoDB, Cassandra, and Riak) for an electronic healthcare system. Lungu and Tudorica [24] developed a benchmark tool for NoSQL systems and compared the performance of MongoDB and MySQL. Rutishauser [25] compared the performance of MongoDB and PostgreSQL using the TPC-H benchmark. Chevalier *et al.* [26] proposed an extension to the Star Schema Benchmark [27] that supports NoSQL systems.

Related to NoSQL data model, Banerjee *et al.* [28] examined whether a semi-structured conceptual model called GOOSSDM can be used as a conceptual model for big data, and Zhao *et al.* [29] modeled MongoDB with relational algebra.

7. EXPERIMENTAL EVALUATION

7.1 Experimental Setup

We compared our method (or CLDA) with the normalization method and the most recent state-of-the-art method (or BFS) [5], which is based on table-level denormalization. The BFS method could produce different denormalized schemas depending on the order of edge visits, and we chose the one with the best query performance. Using the TPC-H benchmark with scale factors (SFs) of 1, 2, 4, 8, and 16, we measured the average query execution time and the accumulated CPU utilization for the TPC-H queries and the database size after migration. For each query, we first ran the query once to warm up the cache and then measured the average execution time for two subsequent runs. We used accumulated CPU utilization to calculate the normalized CPU cost where the denominator for the normalization is the accumulated CPU utilization of CLDA. The databases of the three methods are small enough to fit in the main memory cache of MongoDB for SF = 1 and larger than the cache for SF = 16. We used MongoDB version 3.2.5 and followed the recommended configuration [30]. Because MongoDB does not support SQL, we manually translated SQL queries into MongoDB queries using the aggregation pipeline [4] and manually optimized each MongoDB query. We created indexes on the primary and foreign key fields to support indexed nested-loop joins. All experiments were conducted on a Dell PowerEdge R720 server with two Intel Xeon E5-2620 v2 CPUs, 16 GB of memory, and Samsung 850 PRO 256GB SSDs. The size of main memory cache of MongoDB was 8GB.

7.2 Experimental Results

7.2.1 Query performance for SF = 1

Fig. 7 compares the average query execution time for SF = 1 where the disk I/O cost is not a contributing factor to query performance. CLDA improves query performance by 2.2 times compared with the normalization method because CLDA replaces an expensive join operation with a scan operation. The time complexity of an indexed nested-loop join is $O(n \log m)$ and that of a scan is $O(m)$ where $n(m)$ is the number of tuples in outer (inner) relation and $n < m$. Thus, the CPU cost of the normalization method is higher than

that of CLDA as in Fig. 8. CLDA improves query performance 1.3 times compared with the BFS method because CLDA reduces the disadvantages of table-level denormalization as discussed in Section 5. For example, the BFS method often needs extract operations, which sort one big denormalized collection. The CPU cost of the BFS method is higher than that of CLDA as in Fig. 8.

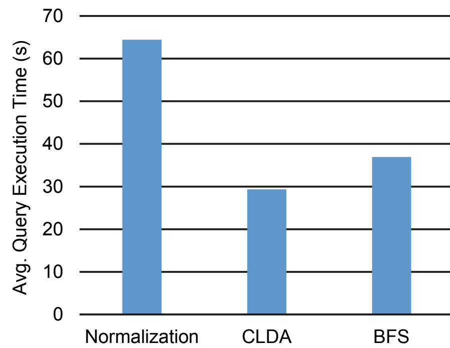

Fig. 7. Query performance for SF = 1.

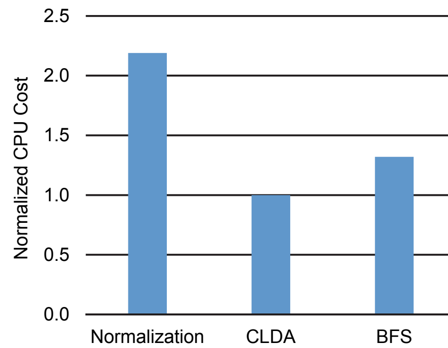

Fig. 8. Normalized CPU cost for SF = 1.

7.2.2 Query performance for SF = 16

Fig. 9 compares the average query execution time for SF = 16 where the disk I/O cost is one of the significant factors. CLDA improves query performance by 1.5 times compared with the normalization method, and 2.0 times compared with the BFS method. Fig. 10 shows the ratio of the database size to the cache size, denoted as *ds*/*cs*. The higher *ds*/*cs* ratio means the more cache misses. The improvement of CLDA compared with the normalization method is decreased from 2.2 times for SF = 1 to 1.5 times for SF = 16 because the *ds*/*cs* ratio of CLDA is higher. We note that, although the *ds*/*cs* ratio of CLDA is higher than that of the normalization method, the CPU cost of CLDA is lower than that of the normalization method as in Fig. 11. The improvement of CLDA compared with the BFS method is increased from 1.3 times for SF = 1 to 2.0 times for SF = 16 because the *ds*/*cs* ratio and the CPU cost of the BFS method are much higher than those of CLDA. The normalized CPU cost of the BFS method for SF = 16 is higher than that for SF = 1 because of the overhead of handling many cache misses.

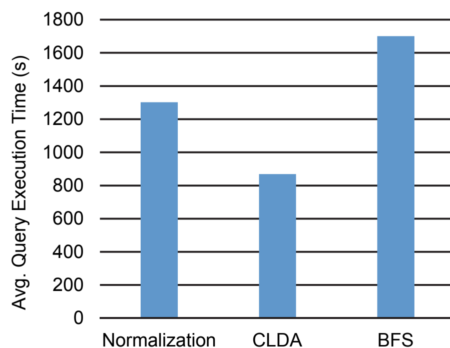

Fig. 9. Query performance for SF = 16.

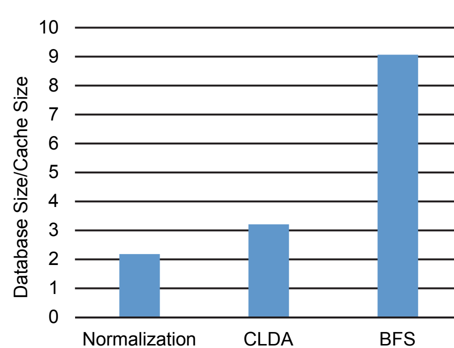

Fig. 10. The *ds*/*cs* ratio for SF = 16.

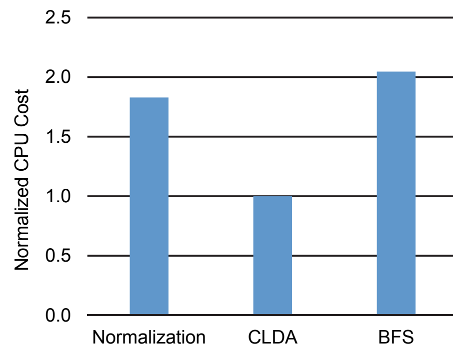

Fig. 11. Normalized CPU cost for SF = 16.

7.2.3 Query performance as SF is varied from 1 to 16

Fig. 12 compares the average query execution time as SF is varied from 1 to 16. Figs. 13 and 14 show the accumulated CPU utilization and the *ds*/*cs* ratio, respectively. CLDA outperforms other methods for all the range of SFs tested because the CPU cost of CLDA is significantly lower than that of other methods as in Fig. 13 and the *ds*/*cs* ratio of CLDA is only a little higher than that of the normalization method as in Fig. 14.

7.2.4 Database size

Fig. 15 compares the database size for SF = 16. Compared with the normalization method, CLDA uses slightly more space (1.5 times), but the BFS method uses much more space (4.2 times). Compared with the BFS method, CLDA uses a 2.8 times less space (65% space savings). The results for SF = 1, 2, 4, and 8 are not shown because they have almost the same pattern.

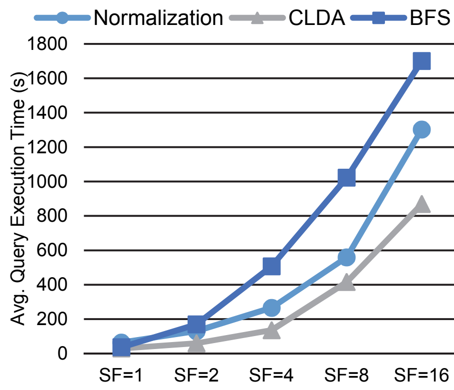

Fig. 12. Query performance for SF = 1 to 16.

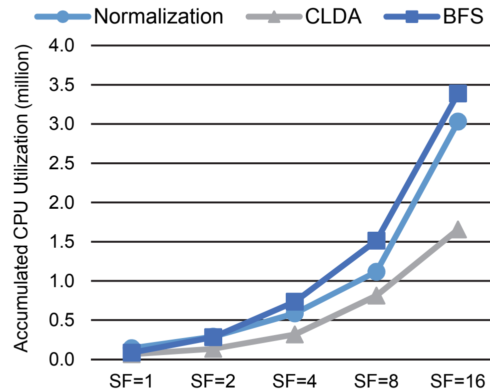

Fig. 13. Accu. CPU utilization for SF = 1 to 16.

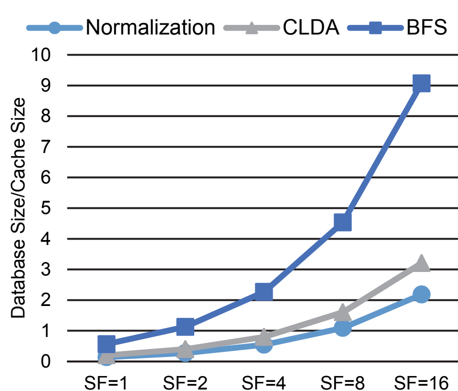

Fig. 14. The *ds*/*cs* ratio for SF = 1 to 16.

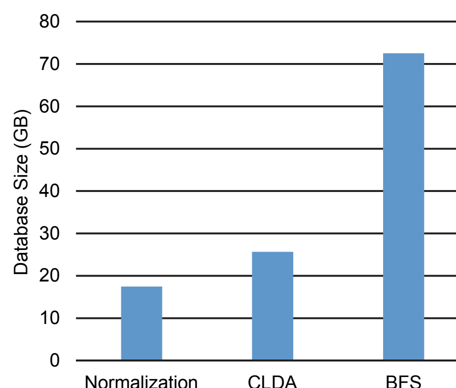

Fig. 15. Database size for SF = 16.

8. CONCLUSIONS

In this paper, we provided a comprehensive solution for database migration from RDBMS to document-oriented NoSQL. Our method reduces both join operations and the disadvantages of table-level denormalization using the notion of column-level denormalization. Furthermore, our method supports atomicity using atomic aggregates. The experimental results show that our method significantly improves the query performance with a marginal increase in the database size compared with the normalization method. Our method also outperforms the state-of-the-art method based on table-level denormalization with a much smaller space.

REFERENCES

1. S. Ombergen, "A comparison of five document-store query languages: finding a suitable standard," Department of Software Engineering, University of Amsterdam, 2014.
2. S. Kim, H. Han, H. Jung, H. Eom, and H. Yeom, "Improving MapReduce performance by exploiting input redundancy," *Journal of Information Science and Engineering*, Vol. 27, 2011, pp. 1137-1152.
3. The TPC-H benchmark, http://www.tpc.org/tpch.
4. The MongoDB 3.2 manual, http://docs.mongodb.org/manual.
5. G. Karnitis and G. Arnicans, "Migration of relational database to document-oriented database: structure denormalization and data transformation," in *Proceedings of International Conference on Computational Intelligence*, *Communication Systems and Network*, 2015, pp. 113-118.
6. DB-engines ranking, http://db-engines.com/en/ranking.
7. P. Li, K. F. Wang, and B. J. Sun, *System for Detecting Migration Differences of a Customized Database Schema*, US Patent 7,496,596, filed Jan. 13, 2009.
8. Y. Huang and T.-J. Luo, "NoSQL database: A scalable, availability, high performance storage for big data," *Lecture Notes in Computer Science*, Vol. 8351, 2014,

pp. 172-183.
9. NoSQL Data Modeling Techniques, https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques.
10. P. J. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, Addison-Wesley Professional, 2012.
11. D. Sullivan, *NoSQL for Mere Mortals*, Addison-Wesley Professional, 2015.
12. G. Zhao, Q. Lin, L. Li, and Z. Li, "Schema conversion model of SQL database to NoSQL," in *Proceedings of the International Conference on P2P*, *Parallel*, *Grid*, *Cloud and Internet Computing*, 2014, pp. 355-362.
13. C.-H. Lee and Y.-L. Zheng, "Automatic SQL-to-NoSQL schema transformation over the MySQL and HBase databases," in *Proceedings of IEEE International Conference on Consumer Electronics*, 2015, pp. 426-427.
14. C. Li, "Transforming relational database into HBase: A case study," in *Proceedings of IEEE International Conference on Software Engineering and Service Sciences*, 2010, pp. 683-687.
15. A. Schram and K. M. Anderson, "MySQL to NoSQL: data modeling challenges in supporting scalability," in *Proceedings of Annual Conference on Systems*, *Programming*, *and Applications: Software for Humanity*, 2012, pp. 191-202.
16. G. Zhao, L. Li, Z. Li, and Q. Lin, "Multiple nested schema of HBase for migration from SQL," in *Proceedings of International Conference on P2P*, *Parallel*, *Grid*, *Cloud and Internet Computing*, 2014, pp. 338-343.
17. C.-H. Lee and Y.-L. Zheng, "SQL-to-NoSQL schema denormalization and migration: a study on content management systems," in *Proceedings of IEEE International Conference on Systems*, *Man*, *and Cybernetics*, 2015, pp. 2022-2026.
18. T. Vajk, P. Feher, K. Fekete, and H. Charaf, "Denormalizing data into schema-free databases," in *Proceedings of IEEE International Conference on Cognitive Infocommunications*, 2013, pp. 747-752.
19. T. Vajk, L. Deak, K. Fekete, and G. Mezei, "Automatic NoSQL schema development: A case study," in *Proceedings of IASTED International Conference on Parallel and Distributed Computing and Networks*, 2013, pp. 656-663.
20. S. K. Shin and G. L. Sanders, "Denormalization strategies for data retrieval from data warehouses," *Decision Support Systems*, Vol. 42, 2006, pp. 267-282.
21. Y. Li and J. M. Patel, "WideTable: an accelerator for analytical data processing," in *Proceedings of the VLDB Endowment*, Vol. 7, 2014, pp. 907-918.
22. A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang, "Can the elephants handle the NoSQL onslaught?" in *Proceedings of the VLDB Endowment*, Vol. 5, 2012, pp. 1712-1723.
23. J. Klein, I. Gorton, N. Ernst, P. Donohoe, K. Pham, and C. Matser, "Performance evaluation of NoSQL databases: a case study," in *Proceedings of ACM/SPEC International Workshop on Performance Analysis of Big Data Systems*, 2015, pp. 5-10.
24. I. Lungu and B. G. Tudorica, "The development of a benchmark tool for NoSQL databases," *Database Systems Journal*, Vol. 4, 2013, pp. 13-20.
25. N. Rutishauser, "TPC-H applied to MongoDB: How a NoSQL database performs," Department of Informatik Vertiefung, University Zurich, 2012.
26. M. Chevalier, M. E. Malki, A. Kopliku, O. Teste, and R. Tournier, "Benchmark for OLAP on NoSQL technologies comparing NoSQL multidimensional data warehous-

ing solutions," in *Proceedings of IEEE International Conference on Research Challenges in Information Science*, 2015, pp. 480-485.
27. P. O'Neil, E. O'Neil, X. Chen, and S. Revilak, "The star schema benchmark and augmented fact table indexing," *Lecture Notes in Computer Science*, Vol. 5895, 2009, pp. 237-252.
28. S. Banerjee, R. Shaw, A. Sarkar, and N. C. Debnath, "Towards logical level design of big data," in *Proceedings of IEEE International Conference on Industrial Informatics*, 2015, pp. 1665-1671.
29. G. Zhao, W. Huang, S. Liang, and Y. Tang, "Modeling MongoDB with relational model," in *Proceedings of the International Conference on Emerging Intelligent Data and Web Technologies*, 2013, pp. 115-121.
30. MongoDB operations best practices, https://www.mongodb.com/collateral/mongodb-operations-best-practices.

JaeJun Yoo has received a B.S. (2000) degree in Computer Science from Soongsil University, M.S. (2002) and Ph.D. (2016) degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST). He is currently a senior research staff at Electronics and Telecommunications Research Institute (ETRI).

Ki-Hoon Lee has received B.S. (2000), M.S. (2002), and Ph.D. (2009) degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST). He is currently an Associate Professor of Department of Computer Engineering at Kwangwoon University.

Young-Ho Jeon has received a B.S. (2015) degree in Computer Engineering from Kwangwoon University. He is currently a candidate for Master of Engineering in Computer Engineering at Kwangwoon University.