AUTOMATIC NOSQL SCHEMA DEVELOPMENT: A CASE STUDY

Tamás Vajk, László Deák, Krisztián Fekete, Gergely Mezei
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Magyar tudósok krt. 2. (Q. épület) QB207
Budapest, Hungary, H-1117
\{tamas.vajk, deak.laszlo, kriszttian.fekete, gmezei\}@aut.bme.hu

ABSTRACT
Cloud service providers offer a huge variety of schema-less NoSQL data storage solutions. The flexibility of these data stores offer greater freedom in structuring the data than relational databases. However, it would be desirable to make use of the strong mathematical background of relational data structures. In this paper, we introduce an automatic NoSQL schema optimization that uses a normalized data schema as starting point. We analyze the predefined set of queries, and compile the schema that can serve the queries with minimal cost at a certain query load. The introduced process is performed on a conceptual model of the database, and the queries are defined in Object Constraint Language to simplify the analysis. The optimization algorithm is introduced through a case study.

KEY WORDS
Cloud computing, Optimization, NoSQL, OCL

1 Introduction

Cloud computing has received significant attention recently. Companies may store their data and perform their computations off-premise in a highly available and scalable environment where they only pay for the resources that they actually use. Compared to traditional infrastructures that only use in-house resources, cloud computing has many advantages that have been transforming the computing solutions used at companies. Naturally, already existing on-premise resources can still be used as part of the infrastructure by connecting them to cloud services and forming a hybrid environment.

Platform as a Service [18] cloud service providers offer several types of data storage mediums. Many of the existing applications store data in relational databases, thus, cloud providers typically implement SQL servers for data storage, which makes the transition to the cloud easier [4]. However, if we need to scale out a relational database and spread across multiple servers, the relational constraints – which the database server can guarantee – are significantly reduced. Referential integrity of the data may have to be checked in application logic, which results in application code that is hard to maintain and performance issues arise.

Thus, cloud providers tend to offer simpler, but more scalable storage options. Google App Engine [17], Microsoft Windows Azure [11], and Amazon Web Services (AWS) [2] all offer different types of NoSQL [22, 7] storage options.

The term NoSQL database is used for a loosely specified class of non-relational data stores. Most of the NoSQL solutions use variations of key-value pairs for storage, where the units of data are sorted and ordered on the basis of the key. Binary Large OBject (BLOB) stores, such as Amazon S3, allow storing any size of binary data identified by a unique key. Also, key-value stores are commonly used in caches. Column-oriented stores, such as Amazon DynamoDB, go one step further, and provide a way to store attribute name and value pairs for each key. A further type of NoSQL solutions is the document database that treats a document as a whole and avoid splitting the document into its constituent name-value pairs. The stored documents encapsulate information in some standard format, such as XML, JSON, which allows for a less rigid data structure than attribute name-value pairs. Document databases, such as CouchDB [3], allow indexing of documents on the basis of not only its key but also its properties (XML tag names, JSON properties).

The theory of relational databases provide precise methods to design schemas with certain properties. Usually, a typical required property is data storage with minimal redundancies to avoid data inconsistencies. However, as NoSQL necessitates less rigidity in the structure of the data, there are no obviously acknowledged ways of structuring data. In this paper, we provide an algorithm that automatically determines the most cost efficient data structuring in a selected subset of NoSQL data stores. Our algorithm works on column-oriented data stores that allows for data sorting and ordering only by the key. Such data stores are Windows Azure Table storage and Amazon DynamoDB. There are column-oriented stores, such as Amazon SimpleDB, in which data can be efficiently selected by any attribute name because the table is automatically indexed by each attribute name. (However, considering a huge amount of data, maintaining the index tables is a performance bottleneck, thus there is the selected group of column-oriented stores, where data is stored in a sorted way by the key and no indexes are allowed.) Our algorithm starts from a predefined relational data schema and a set
of queries, and performs automatic schema denormalization to find the optimal storage scheme regarding a given query load. In our work, SQL query and data definition language could have been used to define the starting point, however, full SQL language support would be a tedious task. Therefore, we have used a metamodeling system, the Visual Modeling and Transformation System [25], that not only allows for simple model definition, but provides a standardized query language, the Object Constraint Language [15], that might be translated to SQL [12] if needed. Having the data model and the queries given in our modeling environment allow us to perform analysis easier. The algorithm is illustrated through a case study. Note that in our solution we only deal with data selection, thus, in this scenario we have not dealt with data consistency. However, eventual consistency could be gained by using materialized views [21], which would slightly modify the cost functions.

The structure of the paper is as follows. Section 2 gives an overview of database performance optimizations and summarizes works that bring SQL knowledge to NoSQL solutions. In Section 3, the case study is introduced, and the steps of the algorithm are illustrated on it. Cost evaluation is also performed in Section 3. Finally, conclusions are drawn and future research options are described in Section 4.

2 Background and Related Work

Performance optimization in relational database systems is a universally emerging task. Oracle provides several options for performance tuning. The Oracle SQL Tuning Advisor analyzes SQL statements and offers tuning recommendations. These recommendations relate to statistics on objects in the database, or advise creation of new indexes, restructuring of the SQL statements, or creating a SQL profile [9, 16]. The optimization can be performed both automatically, in which case the problematic SQL statements are identified and tuned by the database, or manually by selecting single SQL statements.

Another Oracle feature is the SQL Access Advisor [14, 16] that performs similar tasks as our algorithm. The SQL Access Advisor improves performance by recommending the proper set of materialized views, and indexes for a given workload. Naturally, the creation and maintenance of materialized views can be time consuming, and space requirements can be significant. Thus, another component of the SQL Access Advisor also recommends how to optimize materialized views so that they can be fast refreshable. Note however, that this feature does not change the original schema but extends it with additional views. Also, this solution makes heavy use of multiple indexing of the tables and views, thus, even if the algorithm was publicly available, it could not be applied to our scenario.

Database denormalization is a well-known method for achieving performance improvement. In [8], a system that enables transparent database denormalization is introduced. The system keeps a record of the mappings between the denormalized fields and the base fields from which they are derived and if the base fields were to be selected/updated, the new fields are returned/modified. The described system does not carry out the denormalization automatically, instead it hides the given denormalization from the database users by translating internally the received queries. In our work, we need to similarly track which denormalization step has been performed along which navigation in order to be able to rewrite the original queries for the new schema. However, in this paper we do not deal with this small part of the problem.

Works have been carried out to enable SQL functionalities over NoSQL solutions. For instance, in [5] a relational layer over Amazon SimpleDB is introduced. SimpleSQL offers a SQL interface that abstracts any knowledge about data modeling, data persistence and data accessing at SimpleDB. Thus, the system allows for easier migration of existing SQL-based applications to the cloud. Naturally, SQL joins are also handled, and as SimpleDB tables are indexed for each column the solution is viable.

In [19], online transaction processing (OLTP) is enabled on key-value stores that have entity group transactions. Entity groups are collections of data entities that are partitioned together. Thus, the data store saves them physically close to each other, which allows for not resource intensive transactions within the entity group. The authors of [19] state that if the workload and entity groups are designed appropriately, the system can mostly avoid distributed transactions, and scaling out could be performed easier.

Similarly to the mentioned efforts, our work also brings previous SQL knowledge into NoSQL solutions. The main difference from previous relational performance optimizations is that in our case tables define only a single index, which highly restricts data querying options. Also, in our case a data cell could store a list of values (for instance in Amazon DynamoDB), which is not allowed by relational databases. However, in this paper, this difference is not examined.

3 Contributions

In this section the mentioned optimization idea is introduced through a specific example. For the case study we have selected a well-known, still rapidly growing social messaging application, Twitter [23]. We do not have insights into Twitter, however, it serves as a good case study candidate because its functionality is known for everyone, there is publicly available information on the size of the stored data, and it is a social application that does not require ACID (atomicity, consistency, isolation, durability) transaction, thus it qualifies for NoSQL storage.
3.1 Static structure

The basic functionality of Twitter is the following. It allows users to write messages (tweets), these messages can be replies to other ones. It is also possible to submit (retweet) a message that has been previously tweeted by someone else. A message may contain special elements, such as other users’ names, topics, links or pictures. Mentioned user names and topics facilitate searching and linking tweets together, while pictures and links allow connecting to content outside the Twitter environment. However, images – as everything else – need to fit into the 140-character upper length limit of a message, thus, those are also links with different presentation logic. Furthermore, links also need to fit into the length limit, thus, the t.co URL shortener service that provides short (20-character long) URLs and stores additional data is used. One of Twitter’s main functionality is that you can follow users, thus, it is possible to subscribe to someone’s Twitter updates. Finally, one can group users into lists, and these lists can be made public to allow others to subscribe to it.

Although the actual class hierarchy used in the system is not available to the public, we can easily design a UML class diagram that could store all the data for the above functionalities. Our overall static structure of the system is illustrated in Figure 1. We have omitted the properties and methods for easier readability.

In this paper, we only deal with a small segment of this class structure, namely, we are working with Users, Tweets and UserMentions, one of the elements contained in Tweets. Figure 2 depicts the classes with the relations between them, the multiplicities are also illustrated to make relational database design possible.

From the class diagram in Figure 2, one can easily come up with a relational database schema that is in the third normal form (3NF) illustrated in Figure 3.

3.2 Queries

Third normal form database schemas are mostly free of update, insertion, and deletion anomalies. Certain types of 3NF tables are affected by such anomalies, however, in practice 3NF normal forms are considered redundancy free schemas, thus, in this paper we use this as our starting point. Relational databases excel in joining tables for performing complex SQL queries. However, in NoSQL solutions joins are expensive and very slow operations. Thus, the schema needs to be optimized to handle queries without or with minimal joins. If the queries are known in development time, this schema transformation can be performed. To optimize the schema along predefined queries,
we need to express the queries in our modeling environment. The Visual Modeling and Transformation System (VMTS) [25] allows for defining Object Constraint Language (OCL) [15, 26, 24] queries over our database model depicted in Figure 3.

We have defined 9 OCL queries in this case study, these queries can be separated based on the tables they work on: user related, tweet related or mixed queries. All of the queries are defined as static methods because in this way they can be easily transformed to static data access layer methods, however, instance-based methods could have been used as well.

The four queries related to the User table are illustrated in Figure 4. Query 1 retrieves a user by its identifier, actually it selects all users with the given ID from the User table and returns the first element. In C# there is the Single() extension method on collections that would do the same in one step, however, OCL does not define such a construct. Note however, if C# data access layer is based on the name column instead of the ID. This query is semantically valid, because Twitter restricts user names (Twitter handles) to be unique. However, this does not mean that there is an unnecessary column in Twitter, this duality allows users to choose new handles without losing previous messages and without Twitter exposing its internal user identification method. (Obviously, there are drawbacks of having internal and public identification methods separately, which result in multiple blog entries explaining how to properly change a Twitter handle.) Query 3 and 4 are for accessing who is following whom. Query 3 lists user names of those whom the given user follows, while query 4 enumerates user names who follow the selected user. These queries require visiting the Follow table and than for each user the name is selected from the User table.

Three Tweet table-based queries are displayed in Figure 5. Query 5 shows that messages can be requested by their IDs. The message IDs of the replies to a tweet are listed by Query 6, which checks for those tweets that have the ReplyTo attribute set to the given identifier. Finally, Query 7 enumerates the IDs of messages belonging to a specific user.

Two queries that retrieve information from both User and Tweet tables are illustrated in Figure 6. Query 8 collects all the information that is needed for rendering the message as seen on the Twitter website. In our simplified model, this means, that a given tweet needs to be returned and all the mentioned users’ names are collected into a sequence. Together these two pieces of information is wrapped in a tuple to compile one return value for the method. Finally, Query 9 collects the information needed for rendering the user’s main page. For simplification purposes, only the user information, the messages and the number of messages are collected. It could also include the number of followers, etc.

| context User |
| static def : getUser(Id : Integer) : User = User.allInstances \rightarrow select(u | u.Id = Id) \rightarrow asSequence() \rightarrow first() |

| context User |
| static def : getUser(Name : String) : User = User.allInstances \rightarrow select(u | u.Name = Name) \rightarrow asSequence() \rightarrow first() |

| context User |
| static def : getFollowerNames(Id : Integer) : Sequence(String) = Follow.allInstances \rightarrow select(f | f.FollowerId = Id) \rightarrow asSequence() \rightarrow collect(f) |
| static def : getFolloweeNames(Id : Integer) : Sequence(String) = Follow.allInstances \rightarrow select(f | f.FollowerId = Id) \rightarrow asSequence() \rightarrow collect(f) |

| context User |
| static def : getFolloweeNames(Id : Integer) : Sequence(String) = User.allInstances \rightarrow select(u | u.Id = f.FollowerId) \rightarrow asSequence() \rightarrow first().Name |
| static def : getFollowerNames(Id : Integer) : Sequence(String) = User.allInstances \rightarrow select(u | u.Id = f.FollowerId) \rightarrow asSequence() \rightarrow first().Name |

Figure 4. OCL queries for Users

3.3 Denormalization

Having the 3NF schema and all the queries permitted by the system allow NoSQL schema optimization. As mentioned earlier, heavy denormalization is a standard way of increasing query performance in NoSQL table stores. Denormalization is performed – as in relational databases – along the foreign keys. In general, performing a denormalization task along a selected foreign can be separated into the following steps: a) copying new columns to the table at the source of the foreign key; b) introducing the original functional dependencies [10, 1] to the new columns; and c) restoring the referential integrity constraints on the new columns.

A denormalization step is depicted in Figure 7. In the figure, each letter corresponds to a column in a table, each solid line represents a functional dependency, and each dashed line corresponds to a foreign key in the schema. Note that C references D, thus if set theory is considered $C \subseteq D$, which – in database theory – is called inclusion dependency [1, 6]. Furthermore, $E_2$, $F_2$ is a subset of $E$ and $F$ respectively, however, in database systems we cannot represent this with foreign keys, because $D$ and $E$ are not keys.

The denormalization steps described above are a bit vague. Consider column C, which references column A in the same table, depicted in Figure 8. If the previous denormalization task is performed, the referential integrity constraint between $C_2$ and $C$ (dashed-dotted line in Figure 8) would not be created. Thus, step a) of the denormalization task needs to copy the referential integrity constraints that
schema, iterate over all the foreign keys, and denormalize
malized schema options. Thus, we start with the original
between the copied columns as well.

The easiest way of finding the schema that belongs to
the lowest cost requires generating all the possible deno-
malized schema options. Thus, we start with the original
schema, iterate over all the foreign keys, and denormalize


Figure 5. OCL queries for Tweets


Figure 6. OCL queries containing both Users and Tweets


Figure 7. Denormalization performed along a foreign key

go between the copied columns as well.

along them. In the next step, we start from these newly
generated schemas, and perform the same algorithm until
we do not gain any further distinct schemas. Consider the
schema on the left in Figure 7. In the first step, there are
two options: $C \rightarrow D$ and $F \rightarrow G$. If the former one is
chosen, we gain the schema on the right in Figure 7 and there
are three foreign keys to go along, however, if $C \rightarrow D$ is
selected again, that would not result in a different schema.
Naturally, these column equalities have to be checked in
the algorithm. Running the algorithm results in altogether
1, 3, 5, and 6 distinct schemas after selecting 0, 1, 2, and
3 foreign keys respectively. And there are no more foreign
keys to select that would result in further distinct schemas.

If the special case is examined in Figure 8, we find
that selecting $C_2 \rightarrow A$ and $C_2 \rightarrow C$ results in different
schemas, which means – as $C_2 \rightarrow C$ is not a foreign key,
but an inclusion dependency – that our algorithm should
perform the denormalization along inclusion dependencies
rather than foreign keys. Furthermore, the special case also
shows that if there is a (one-long or longer) loop in the
schema, it is not possible to generate all the denormalized
schemas, because there are infinitely many of them. How-
ever, it is possible to find a reasonable limit on the depth of
the solution space, as it cannot be deeper than the longest
navigation in the predefined queries. Thus, if there are no
recursive calls, it is possible to determine a depth limit.

In the Twitter example, the number of distinct
schemas until a given level are the following: 1, 7, 27, 84,
253, 821, 3097, 14176, 78894. In our case, Query 9 con-
tains the longest path, it starts in the User table, visits the
Tweet table multiple times, goes through the UserMention
table to reach the user name in the User table. Thus, the
path uses three navigations, which means that 84 schema
variations need to be analyzed in this case study.

3.4 Query answerability

In Section 3.3 we have generated all the schemas that need
to be examined for finding the optimal solution. Each query
defines a navigation path that is needed to be followed dur-
ing query evaluation. For our algorithm, the queries are
simplified into 3-tuples containing the following pieces of
information: $a$) the attribute that is used as the parame-
ter of the query; $b$) the list of foreign keys that are used
for navigation in the query; and $c$) the attributes that are
returned by the query. To determine how a given query
can be answered on a selected schema, we have to select exactly one attribute in each table that should be indexed. For instance, consider the initial schema in Figure 3 without the primary keys and Query 1. Query 1 is simplified into \( User[Name], \{\}, \{ User[Id], User[Name], \ldots \} \) 3-tuple, meaning that it starts from the \( Name \) attribute, does not need any navigations along foreign keys, and returns all the attributes in the \( User \) table. In our schema this would mean, that the index should be placed onto the \( Name \) attribute. Also as all the attributes needed to be returned are found in the table, this schema can answer the selected query.

Another – a bit more complicated – example can be gained on the same schema by considering Query 3 (\( getFollowerNames() \)). In this case, the query is compiled into the \( (User[Id], \{FK2, FK1\}, \{User[Name]\}) \) 3-tuple. In the schema this would mean that the index in the \( User \) table should be placed onto the \( Id \) attribute, in the \( Follow \) table \( FolloweId \) columns should be indexed, and luckily the \( User \) table is queried by the \( Id \) attribute again, which does not conflict with the previously selected indexed column.

Note that query answerability does not only mean selecting the indexed column, there are cases when further schema modification is necessary. In the previous examples there were no conflicting index selection scenarios, however these kind of inconsistencies can emerge as having only one indexable column means a severe restriction. Consider Query 1 and Query 2 simultaneously on the initial schema. The former one would require the \( Id \) to be indexed, the latter one the \( Name \), which contradict. Thus, to overcome this issue, we have to modify the schema in some way. In our work, we only consider the following two options: a) duplicate the entire table that is in a conflicting state and use different indices; or b) create an index table [13]. Note that both options are viable and both should be considered as their costs are different under different query usage patterns as seen in Section 3.5. Thus, for Query 1 and 2, the two options would result in either \( User(Id, Name, \ldots) \) and \( User'(Id, Name, \ldots) \) or \( User(Id, Name, \ldots) \) and \( User'(Name, Id) \) schemas, where the underlined attributes are the indexed ones. In case of the former option, querying both Query 1 and 2 needs only one request to the server, while in the latter option, Query 2 can only be answered by two requests. However, the second option requires considerable less storage space if there are lots of columns in the \( User \) table.

Finally, it is important to note, that the query order in which we process the predefined queries may result in different schemas, thus, all the permutations should be checked. To demonstrate this, consider Query 1 and 2 from the previous example. If Query 1 is analyzed before Query 2, we gain the schema options described above. However, if the other order is considered, the index table version becomes the following: \( User(Name, Id, \ldots) \) and \( User'(Id, Name) \), which might only seem subtly different, but may result in huge cost differences if there is heavy user selection by \( Id \) and light by \( Name \).

3.5 Cost function

Section 3.3 and 3.4 illustrated how all the schemas have been generated. The final step in finding the optimal schema to our problem is calculating the aggregate cost of the storage and prospective query load. In Windows Azure [11], storage is billed based on the amount of data stored and the number of storage transactions (queries). Amazon DynamoDB [2] pricing is also based on the amount of data, but the other component is based on provisioned throughput capacity, which makes comparing the two services difficult. This section is mainly concerned with the cost prediction of Windows Azure, however, at the end of the section we draw some comparison between the two service providers.

Consider our simplest example, which would only contain Query 1 and 2. This results in three distinct schemas: a) duplicate tables; b) index table for Q1; and c) index table for Q2. The storage capacity costs \( c_S S_{Col} N_{Col} N_{Row} \) for a) and \( c_S S_{Col}(N_{Col} + 2) N_{Row} \) for b) and c), where \( c_S \) is the price in USD for each stored GB of data. \( S_{Col} \) is the average size of a cell in GB, \( N_{Col} \) is the number of columns, and \( N_{Row} \) is the number of rows in the table. The query cost is \( c_T(N_{Q1} + N_{Q2}) \) for option a), \( c_T(2N_{Q1} + N_{Q2}) \) for option b), and \( c_T(N_{Q1} + 2N_{Q2}) \) for option c), where \( c_T \) is the transaction cost, \( N_{Q1} \) and \( N_{Q2} \) are the number of times Query 1 and 2 are executed respectively. From the Twitter API, we could estimate that \( N_{Col} \) is around 20, and \( S_{Col} \) should not be considerably more than 10 bytes, as most of the fields are integers, there are only a few strings. Also, the number of users in Twitter is around 140 million [27]. We have used $0.01/100000 as \( c_T \) and $0.1/GB as \( c_S \). These leave only \( N_{Q1} \) and \( N_{Q2} \) as free variables. Figure 9 depicts the costs of the three different schemas in the function of the executed query numbers. It can be concluded that above a certain query number, the price of transactions overtake the price of the storage, thus, duplicating the user table is financially beneficial.

If one loads someone’s Twitter page, the basic

![Figure 9. Costs of different schemas with different usage patterns](image-url)
information of the user and their tweets are displayed. In our case study this is modeled by Query 9, thus, let us calculate the cost of serving this query from a highly denormalized and a highly normalized data schema. Our original schema displayed in Figure 3 will suffice as a normalized schema, fortunately there are no index collisions, as if the Id of the User table, the UserId of the Tweet table, and the TweetId of the UserMention table are indexed, it is possible to answer the query. As the highly denormalized table, consider the denormalization along FK2 in UserMention, FK1 in UserMention and FK1 in the Tweet table. This would result in the following relation: R(UserId, UserName, User..., TweetId, TweetText, Tweet..., UserMention, User_Name), where we have renamed the attributes for easier readability. Using the previous notations, Equation 1 displays the requires storage space for the normalized schema.

\[ N_{U\text{Col}} S_{U\text{Col}} N_{U\text{Row}} + (N_{T\text{Col}} + 1) S_{T\text{Col}} N_{T\text{Row}} + 3 S_{U\text{MCal}} N_{U\text{MCal}} \]  

(1)

The required query number is 4, as we need to select the user, then the tweets belonging to him, after that the mentions, and finally, the names for the user mentions. On the other hand, the denormalized schema requires only 1 query, but consumes much more space. The required space can be estimated with Equation 2.

\[ N_{U\text{Col}} S_{U\text{Col}} N_{U\text{MCol}} + N_{T\text{Col}} S_{T\text{Col}} N_{T\text{MCol}} + S_{U\text{MCal}} N_{U\text{MCal}} \]  

(2)

To visualize the above two, we might use the previous assumption about the system. Furthermore, we deliberately take \( N_{T\text{Col}} \) as 15, \( S_{T\text{Col}} \) and \( S_{U\text{MCal}} \) as 10 bytes, let \( N_{T\text{Row}} \) be 14 billion, which would mean 100 tweets per user on average, and for simplicity let \( N_{U\text{MRow}} \) be 14 billion as well. In this case, the denormalized data would take up about 4700GB of storage space, while the normalized one is about 2500GB. Note that according to [27] there are 340 million new tweets each day, which does not directly correlate to the number of queries, but calculating with this number and assuming 100 tweets per user on average means 3.4 million Query 9 requests per day, which is about 100 million per month. But in this calculation we have used the number of insertions, which should be significantly less than the select queries, thus we can assume that the number of queries is way beyond the tipping point at 700 million shown in Figure 10.

Note that we have not taken into account that in Windows Azure a single query can return only 1000 rows and a continuation token, which may be used to query more rows for the same query.

As mentioned before Amazon uses a different charging method, prices are based on the provisioned throughput of the system. For 50 reads per second, the customer is charged $0.0113 per hour. In [20], 70000 API calls per second is mentioned as the load on the Twitter servers. Let us calculate with this number, although it is not simply for selections and not up-to-date, it would result in $11500 per month plus $5300 for 4700GB of storage. In DynamoDB, Amazon uses SSD drives, which raises the storage price considerably. Altogether the calculation results in $16800. While using the same numbers in Windows Azure, and considering a constant load on the servers, 70000 queries per second means 184 billion queries per month. In Windows Azure this would cost $18400 and $500 for storage. From this estimation Windows Azure seems more expensive, but note that in the calculation we have been using the maximum load that our provisioned throughput could serve in the Amazon solution.

4 Conclusion

With the proliferation of cloud services in industrial applications, an increasing number of companies migrate their applications into the cloud. The immense amount of data used in these applications require highly distributable data stores, such as schema-less NoSQL tables. It would be desirable to leverage the previously gained schema design knowledge in these new data stores. Thus, in this paper, we have illustrated a method that automatically transforms the predefined Third Normal Form database schema into a NoSQL schema. On this resulting schema, the predefined set of queries can be executed, and the storage and operation costs of the new schema is minimized along a certain query load.

The optimization is performed by denormalization, which could be further improved. In the future, we aim at handling aggregate functions, such as count(), which should not be handled by querying the data and then completing the aggregation in memory, but by storing the aggregated value denormalized in the data store.

Acknowledgements

This work is connected to the scientific program of the “Development of quality-oriented and harmonized R+D+I strategy and functional model at BME” project, sup-
ported by the New Hungary Development Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

References


