Modeling Strategies for Storing Data in Distributed Heterogeneous NoSQL Databases

Moditha Hewasinghage, Nacéra Bennacer Seghouani, Francesca Bugiotti

Modeling Strategies for Storing Data in Distributed Heterogeneous NoSQL Databases

Moditha Hewasinghage[1]*, Nacéra Bennacer Seghouani[2], Francesca Bugiotti[2]
[1]Universitat Politcnica de Catalunya, [2]LRI, CentraleSupélec, Paris-Saclay University

Abstract. In this work, we propose HerM (Heterogeneous Distributed Model), a NoSQL data modeling approach which supports the use of multiple heterogeneous NoSQL systems in a distributed environment. We define the conceptual elements necessary for data modeling, and we identify optimized data distribution patterns. We implemented a flexible framework, where we deployed our proposed modeling strategies and that we evaluated comparing our approach against native the NoSQL data distribution methodology provided by the NoSQL databases MongoDB.

Keywords: Conceptual Modeling, Heterogeneity, Data Distribution

1 Introduction

How to store and use Big Data to extract information efficiently and effectively has become crucial in many fields. To support this, the industry has started moving towards new systems called non-relational or NoSQL. These systems are highly scalable, can efficiently handle large amounts of data, and support flexible, semi-structured data. NoSQL systems are mostly based on simple read-write operations where performance is a crucial concern and where ACID properties can be traded off [6, 11]. Currently, there are more than one hundred NoSQL systems available with different characteristics of the data model, different data access APIs, and different consistency and durability guarantees [14]. The various data models have been categorized into four main categories namely the key-value, the document, the column-family (or extensible record), and the graph [6]. This vast heterogeneity has opened up new problems. NoSQL data-stores are claimed to be flexible, but research [1, 4, 5] has shown that even in the NoSQL world data design requires significant modeling decisions that impact on considerable quality requirements. Moreover a complete and fully formalized data modeling procedure for NoSQL systems does not exist. The distribution of data plays a significant role in NoSQL systems [6] and many research approaches have introduced efficient methods and algorithms for distributing data [3, 9]. In this paper we define modeling strategies that support the usage of multiple heterogeneous data-stores in a distributed environment. Our approach enables modeling heterogeneous data stores and provides a configurable data distribution methodology, achieved thanks to the usage of a general conceptual model composed by a set of general constructs. We also identify data distribution strategies at the

* This work was achieved at LRI Paris-Saclay University.

conceptual level. Our approach provides a flexible mechanism to deploy the conceptual model that is implemented through a framework that allows the users access data efficiently and transparent of the underlying data stores. We based our data model, called HerM (**He**terogeneous Dist**r**ibuted **M**odel) on an existing general data model for heterogeneous NoSQL systems: the SOS Model [2]. HerM extends SOS to support distributed environment and can be used to evaluate the best data distribution strategy for different use cases. We map HerM into a logical model with a step by step methodology, mapping the relations and distributing the data among the physical cluster nodes. We use MongoDB and Oracle NoSQL as two heterogeneous data models. Next, we implement a flexible framework that provides transparent access to the data with a familiar REST API and a JSON based, easy-to-learn query interface. We offer the ability in configuring the data model for the system through simple configuration based on HerM giving much more portability and flexibility in supporting different use cases. Finally, we evaluate our framework against a native sharded MongoDB instance on a large dataset based on Twitter, for different data storage and retrieval scenarios. With this comparison, we show that HerM provides useful distribution strategies and that implemented model performs on par with native MongoDB. Moreover, the system is exceptionally stable and give constant performance throughout the experiments.

This paper is organized as follows: in Section 2 we explore the related work, then we define our data model in Section 3. The framework we implemented is introduced in Section 4. We finally describe our experiments in Section 5.

2 Related Work

Only a few research has been conducted on systematically modeling NoSQL data-stores. In [2] the authors introduce a high-level description of an interface for NoSQL systems based on a generic model. They propose a meta-modeling approach that inspired our research where a common underlying structure is defined with the methods to access the system. [5] introduces a NoSQL abstract model called NoAM which is designed to support scalability, performance, and consistency but does not take into account the data distribution process. Another general approach for designing a NoSQL system focused on analytical workloads have been discussed in [7]. Different works have analyzed the problem of selecting a target cloud data-store. Among those [10] defines a general XML schema that proposes a multi-criteria optimization model for calculating the optimal data distribution. Some other works have addressed the problem of querying data without proposing a specific general data model. DBMS+ is a new notion introduced in [8] where a DBMS encapsulates the different execution and storage engines. OPEN-PaaS-DataBase API (ODBAPI) [12] defines a unified REST-based API for executing queries over NoSQL data-stores. This approach is extended to a concept of a virtual data-store which enables complex queries over heterogeneous data-stores [13]. Finally ESTOCADA [4] is a constraint-based query rewriting system for heterogeneous NoSQL data-stores

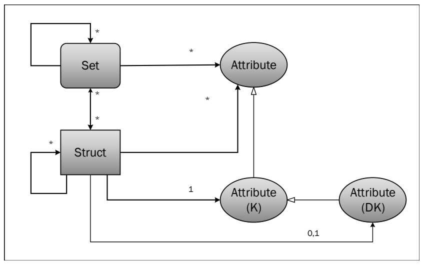

Fig. 1: HerM Meta Model

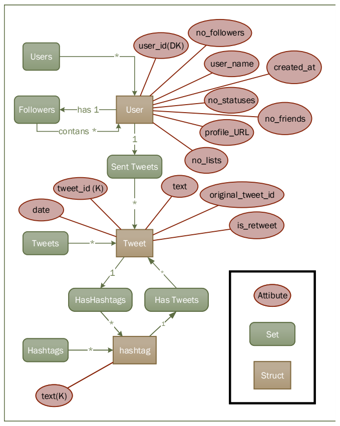

Fig. 2: HerM model of the Twitter example

that chooses the optimal data-storage methodology for the required scenario and the queries are executed depending on how they are interacting with the different data-stores.

3 Heterogeneous Distributed Model (HerM)

In this section, we discuss our general distribution modeling strategy on heterogeneous NoSQL systems. This modeling strategy is based on the definition and on the usage of HerM (**He**terogeneous Dist**r**ibuted **M**odel), a general conceptual data model.

3.1 HeRM data model

The starting point of our conceptual model is the SOS data model [2]. The SOS data model can represent the main categories of NoSQL systems (key-value, documents, and column) but does not offer data distribution facilities. Therefore, we extend it and propose HerM. SOS data model defines three major constructs: *Attribute*, *Struct*, and *Set*. Simple elements of entities such as key-value pairs can be modeled as *Attributes* and groups of heterogeneous *Attributes* can be represented as a *Struct*. A collection of entities are represented in a *Set*. A name and an associated value characterize each construct. The structure of the value depends on the type of the construct. An *Attribute* can contain a simple value such as an integer or string. *Struct*s and *Set*s are complex elements which can contain multiple *Attributes*, *Structs*, *Set*s or combination of those. Different kind of binary relations can relate entities, but many-to-many relations are not taken into account. A database is a set of collections of these constructs. Our data distribution approach considers the conceptual model as a hierarchy of related entities. This hierarchy is mapped into a data distribution strategy. The most important entity of the hierarchical representation is the *Root* that follows this rule: “The *Root Struct* should not have any direct or indirect incoming connections from another *Struct* with many-to-many cardinality.” An incoming relation for an entity means that the entity is nested into the referring one.

In Figure 2 shows an example of the use of our data model for storing Twitter data. Three entities that are candidate to be elected as the *Root* entity: User, Tweet, and hashtag. In the given example the only *Struct* that satisfies this requirement is the user. If there are multiple candidates for the *Root* the end user could choose one of them and have the others as child entities. The possibility of having more than one distribution is beyond our scope. The entities are represented as *Struct*s in the model. In the example, it is the user_id. All the other *Struct*s that represent an entity should have an *Attribute* that helps to identify the entity uniquely, referred as Key (K).

The *Distribution Key* (DK) denotes the distribution field. The data is stored in the different available physical nodes of a NoSQL distributed data-store handled by the framework according to the DK. To guarantee an even and correct data distribution, the distribution key must be chosen among the candidate attributes of the *Root*. The HerM meta model is shown in Figure 1.

3.2 From HerM to the Logical Model

After identifying the *Distribution Key* of the *Root* and the keys for other *Struct*s the next step is to determine the collections of our entities.

Identifying the collections Collections are the independent sets in HerM model. An independent set is a *Set* which does not have any incoming connections and is directly related to a *Struct*. In Figure 2 there are three independent *Set*s Users, Tweets, and Hashtags.

Data Distribution In our approach, the data distribution mechanism is guided by the Distribution Key. The different parts of data lie on different data-stores as well as different physical nodes. Data should be distributed *in a logical way*: data locality is important and minimizes the joins and the data duplication.

Storing different entities in different data-store collections allow effective data manipulation concerning the usage of a single data-store. The root entity is the main entry point of the data distribution. The distribution key defines where the root entity instances will lie on the physical node. It is impossible to have a perfect distribution mainly because the related entities are of different sizes for a root entity instance. According to the hierarchy fixed from the data model, the entities related to the *Root* entity are stored in the same physical node. In HerM model an entity is represented as a *Struct*. The relationships between the entities are expressed differently depending on the cardinality between the structs that is specified in the data model. A *One-to-one* relationship relates two structs, one of the *Struct*s could be chosen to contain the attributes of the other *Struct* into it. This merging process will then result in a single *Struct* having all the attributes of both the *Struct*s. A *One-to-many* relationship relates a *Struct* and to a *Set* of another *Struct*. To represent this relationship, the *Struct* will contain a new *Set* with all the key attributes of the *Struct*s contained in the nested *Set*. All the *Struct*s contained into the nested set will have a new attribute holding the key of the father *Struct* (allowing the reverse lookup). A *Many-to-many* relationship relates two *Set*s of *Struct*s. To represent this relationship, the two *Struct*s will contain a new *Set* with all the key attributes allowing a bidirectional lookup. In this case, it is not guaranteed that the nested entities would lie on the same physical node. Therefore, both many relationship entities will be distributed among different nodes, and one will be repeated.

4 HerM Framework

The overall architecture of the HerM Framework is shown in Figure 3. It is a master-slave architecture and consists of one main physical node and multiple physical data nodes. The main node is responsible for managing the overall data distribution, handling the queries, and returning the results. Each data node contains the NoSQL data-store instances which are responsible for the data storage and managing the queries (and the joins if necessary). The framework is able to take into account the following aspects: (*i*) Providing a transparent query interface; (*ii*) Dynamically modeling the meta-layer; (*iii*) Handling the distribution of a single entity and the joins among related entities. The meta-layer plays a

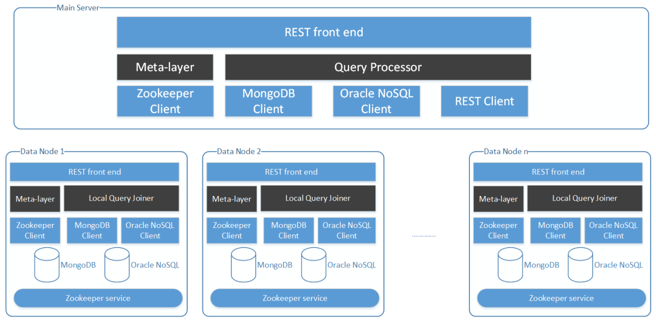

Fig. 3: Overall architecture of HerM Framework

significant role in HerM framework as it contains all the necessary information about the relationships and the key information of the entities defined in the meta-model. It keeps the name of the entity, whether it is the root entity, the parent entity (if it is not the root entity), primary key of the entity, reverse index to the parent entity (if it is a child entity), the type of the relationship with the parent (one or many) for all the entities. This information helps to identify and execute queries throughout the HerM Framework. It also carries the information related to the physical data distribution and the target data-store parameters. This information needs to be available and updated throughout all the data nodes. Therefore we used Apache Zookeper which helps to manage configuration information in a distributed environment. The meta-data is stored in JSON format, and the individual data nodes have their own Zookeeper instance.

Handling data distribution and joins of related entities The data distribution is handled in two steps. The first one consists of the initial distribution of the root entity among the data nodes and the second one is the distribution of the related entities stored in separate data stores according to the distribution key. In our example the USER_ID is selected as the distribution key, and the user's data is split among the available number of data nodes with ranges. When insertion of a USER is sent as a query the HerM Framework analyses the data and determine that it is the root element then according to the distribution key USER_ID defines the data node for the data as well as the datastore and the collection. Handling joins, or related entities is a challenging task since most of the NoSQL systems do not provide the join operation. Moreover, HerM framework uses different data stores, which means that there are no pre-existing tools that support this. We propose to execute the joins locally on the data node where possible. By doing this, we expect the system to be more stable and the workload on the central location to be less, resulting in better overall performance. The performance also increases because our framework distributes data trying to put related entities in the same physical node.

5 Experiments

In this section, we describe the different experiments we run to evaluate the performances of HerM Framework. We studied different INSERT and SELECT queries and compared to MongoDB, analyzing the performance and the stability. The experiments were carried out in a cluster environment using MongoDB 3.4 and Oracle NoSQL 4.3.11. The data was collected from Twitter using the Twitter REST API containing 412814 users 275139 tweets and 8203 unique hashtags that occur in these tweets. The data was distributed among four data nodes using the distribution of the USER_ID as distribution key as described in Figure 2. The quartiles of the USER_ID were used to distribute the data. The users were stored in MongoDB and the tweets of the user are stored in the Oracle NoSQL instance in the same data node as well as the HASHTAGS. The main HerM Framework instance was running on a separate node. The same nodes were used as the MongoDB shard instances. USER is stored as the main collection of documents with embedded TWEETS and HASHTAGS.

The experiments evaluated four different scenarios: *(i)-(ii)* The system and MongoDB without indexes, *(iii)* The system with an index on USER_ID in MongoDB and *(iv)* native MongoDB with index on USER_ID and TWEET_ID.

Table 1 shows the runtime in milliseconds for inserting users, tweets, querying a user, tweet, and followers of a given user. The average insertion time of USER in native MongoDB is faster compared to the system in both with and without indexes. This is expected because the introduction of the REST API introduces additional overhead.

Query/ System	INSERT user	SELECT user	INSERT tweet	SELECT tweet	SELECT followers
HerM Framwork	1.45	80.05	4.11	31.1	147.92
Native MongoDB	1.22	108.90	101.33	134.74	107.84
HerM Framwork + index	2.14	33.82	4.13	29.80	39.10
Native MongoDB + index	1.43	12.64	10.10	21.70	10.76

Table 1: Average query run times

The performance of the system for querying users is nearly 30% faster than querying without indexes the native MongoDB shard instance. Querying without an index produces a full scan on the documents and in our system we have to do the scan only on the relevant node where MongoDB has to do a scan on all the nodes. Since the tweets are stored as nested elements inside the user document in MongoDB, in each insert a push operation to the nested collection is needed. Indexing on native MongoDB improved the insertion time by improving the finding of user. The next experiment was to query a tweet by the ID. In our system, the reverse index for the USER_ID, TWEET_ID stored in Oracle NoSQL is referred to make the major-minor key combination to retrieve a particular tweet. In MongoDB the tweets are stored as nested objects making retrieval of an inner element costly. When retrieving the followers, in our system, the user is retrieved at first. Once the USER_IDs of the followers are retrieved from the user, the IDs are divided between the nodes as it is the distribution key. Then separate nodes will retrieve a set of USER_IDs that lies in the node. Finally, the users from all the nodes are merged. In native MongoDB, the user followers are retrieved directly given a list of ids. MongoDB shard with the indexes has the best performance closely followed by our system with the index. We also measured the percentage of outliers using Tukey's method for each of the scenarios in both systems. For inserting a user, the percentage of outliers is less than 5% which can be considered as usual. In the inserts for MongoDB, we are inserting close to 300000 tweets, and almost 14% of outliers mean that around 42000 of them have taken quite more than the average time and the maximum value was about 3 seconds for an insertion. This performance issue could be a result of the complex nested structure of the data model. The results of the experiments show that the performance and the stability of the system are better compared to the native MongoDB almost all scenarios. The performance measured in the above experiments are based on the total time taken from the moment the client sends the REST request to the API until the server sends the response. This total time contains certain additional overheads which are beyond our control. Mainly the REST client and the server implementations in jersey. Figure 4 shows the

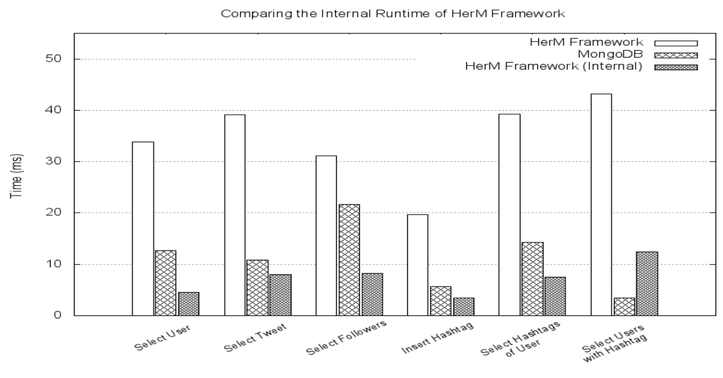

Fig. 4: Internal average query time comparison

internal average query runtime of the system compared to the total time of the system with indexes and the average time taken by the native MongoDB cluster with indexes for different queries. The internal time of the system is quite low in all the cases. In almost all the cases the difference between the total time and the internal time has a difference of around 25-30 milliseconds. This constant time is for the additional requirements added by the REST API: the time to create the REST client by the end user, create the request, time for the request to reach the server, server processing the request, time for the server to serialize the response and the time for the response to reach the server. The results shows that the average internal time is not only better than the overall time but also better than the native MongoDB shard as well.

Our experiments show that our modeling approach has better or comparable performance with respect to the native MongoDB implementation. The performance of the REST API is beyond our control: we define our framework without relying on third party software or optimizing the implementation of the APIs. Given that the aim of this paper is to show that our approach of data modeling in NoSQL system has its advantages those factors are beyond our scope.

References

1. Atzeni, P., Bellomarini, L., Bugiotti, F., Celli, F., Gianforme, G.: A runtime approach to model-generic translation of schema and data. Inf. Syst. **37**(3), 269–287 (2012)
2. Atzeni, P., Bugiotti, F., Rossi, L.: Uniform access to nosql systems. Inf. Syst. **43**, 117–133 (2014)
3. Basani, V.R., Mangiapudi, K., Murach, L.M., Karge, L.R., Revsin, V.S., Bestavros, A., Crovella, M.E., LaRosa, D.J.: Method and apparatus for reliable and scalable distribution of data files in distributed networks (Apr 6 2004), uS Patent 6,718,361
4. Bugiotti, F., Bursztyn, D., Deutsch, A., Manolescu, I., Zampetakis, S.: Flexible hybrid stores: Constraint-based rewriting to the rescue. In: 32nd IEEE International Conference on Data Engineering. pp. 1394–1397 (2016)
5. Bugiotti, F., Cabibbo, L., Atzeni, P., Torlone, R.: Database design for nosql systems. In: ER. pp. 223–231 (2014)
6. Cattell, R.: Scalable SQL and NoSQL data stores. SIGMOD Record **39**(4), 12–27 (2010)
7. Herrero, V., Abelló, A., Romero, O.: Nosql design for analytical workloads: Variability matters. In: ER. pp. 50–64. Springer (2016)
8. Lim, H., Han, Y., Babu, S.: How to fit when no one size fits. In: The Conference on Innovative Data Systems Research (CIDR) (2013)
9. Pokorny, J.: Nosql databases: a step to database scalability in web environment. International Journal of Web Information Systems **9**(1), 69–82 (2013)
10. Ruiz-Alvarez, A., Humphrey, M.: A model and decision procedure for data storage in cloud computing. In: 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. pp. 572–579. IEEE (2012)
11. Sadalage, P.J., Fowler, M.J.: NoSQL Distilled. Addison-Wesley (2012)
12. Sellami, R., Bhiri, S., Defude, B.: ODBAPI: a unified REST API for relational and NoSQL data stores. In: IEEE International Congress on Big Data. pp. 653–660 (2014)
13. Sellami, R., Bhiri, S., Defude, B.: Supporting multi data stores applications in cloud environments. IEEE Transactions on Services Computing **9**(1), 59–71 (2016)
14. Stonebraker, M.: Stonebraker on NoSQL and enterprises. Commun. ACM pp. 10–11 (2011)