2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing

Issues in Handling Complex Data Structures with NoSQL databases

Santo Lombardo, Elisabetta Di Nitto and Danilo Ardagna
Dipartimento di Elettronica e Informazione - DEI
Politecnico di Milano
Piazza Leonardo Da Vinci, 32 - 20133 - Milano
Email: {lombardo, dinitto, ardagna}@elet.polimi.it

***Abstract*—The emerging storage systems for Cloud Computing are so-called NoSQL databases. These systems lack the join mechanism and declarative queries approach but show high scalability and performance. While most famous ICT companies have no doubts about the usage of NoSQL, there is still a lack of methodologies and mechanisms to support the design and management of a NoSQL database starting from a complex high level data schema. The final goal of our work is to equip NoSQL databases with programming abstractions that make them close to the expressiveness of RDBMS, still maintaining their significant advantages in terms of scalability. In this paper we discuss the main shortcomings of using a NoSQL approach and we show how we plan to overcome them.**

I. Introduction

With the growing popularity of the Cloud, the way of designing applications is changing radically. The Cloud forces a design approach that makes the front-end and back-end strongly decoupled. Furthermore, the ability of the Cloud to support scalability requires special care in the design of the back-end system. In particular, the data storage solution chosen by the developer can have a big impact for what concerns scalability. In particular, if a relational database management system (RDBMS) is adopted, it could become a bottleneck for the application [1].

For this reason, new storage systems known as NoSQL DBs are increasingly used and proposed by major ICT companies. In [2] Facebook team discusses about its experience and the decision to migrate parts of its system to Cassandra. Major Cloud providers today offer a NoSQL service (i.e. Azure [3], Amazon [4], [5], Google [6], etc.). Moreover, Apache has developed Hadoop [7], a distributed map-reduce framework including alsa a NoSQL implementation.

NoSQL DBs are usually based on simple tables. Tables host all data and do not have correlation with each other. Users are provided with a small set of primitive operations to manipulate and retrieve data. This simplifies the use for novice developers but requires significant amount of programming work to develop complex business logic.

The goal of our work is to equip NoSQL DBs with powerful programming abstractions that make them closer to the RDBMS expressiveness, still maintaining the significant advantages the NoSQL approach offers in terms of scalability. In the following of this paper we present a realistic use case (Section II) and exploit it to highlight the difficulties that can be encountered in the development of a relatively complex system based on a NoSQL DB (Section III). Moreover, we identify the main principles on top of which we are building a data design and management framework, Marijà, with the aim of addressing these problems (Section IV). Finally, we present some related literature proposals (Section V) and conclude providing an outlook on future work (Section VI).

II. Use case: ACME

In this section we introduce a case study that will be used in the reminder of the paper.

ACME is a small start-up company aiming at publishing, with a very limited budget, a new on-line service. ACME is attracted by the most exciting features of the Cloud, in particular, by its capability to dynamically adapt to workload fluctuations. Thus, ACME decides to build its solution on a Cloud system. This way, the company will pay only for the real usage of resources and will be able to scale up the system based on the growth of users.

The ACME service is focused on supporting the owners of some business (e.g., a gym, a theatre, a small shop), which we will call *providers*, to publish offers (*bouquets* in our terminology) for their potential *customers*. For example, a bouquet could be a 30 entrances ticket for the gym. Bouquets can be subscribed by customers. A *credit* is associated to each subscription (e.g., 30 is the initial credit in the case of the gym ticket bouquet) together with a *policy* that defines the way the credit should be modified based on the actions of the customer. For instance, whenever the customer goes to the gym, the credit should be decreased by one. In this case, the *policy* is `decrease by one`. ACME helps providers in managing bouquet in a simple way, keeping track of the registered customers and of the operations they perform. Figure 1 describes the data model of the ACME service using an entity-relationship diagram. In the diagram `P`, `B` and `C` entities represent providers, bouquet and customers, respectively. `R1` represents the *hasBouquet* relationship that relates a provider with a bouquet. As the cardinality shows, the same bouquet can be offered by many providers and a provider can offer many bouquets. `R2` represents the relationship *isPreferred* that associates one provider to exactly one bouquet, the one it prefers to offer. Finally, `R3` represents the *hasSubscribed*

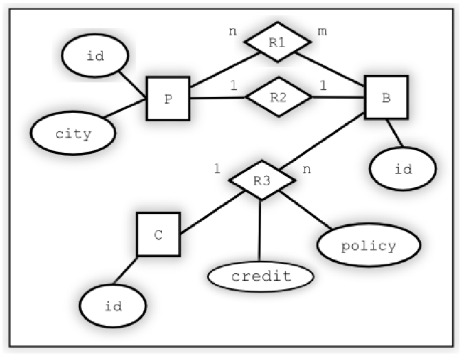

Fig. 1. ACME data model.

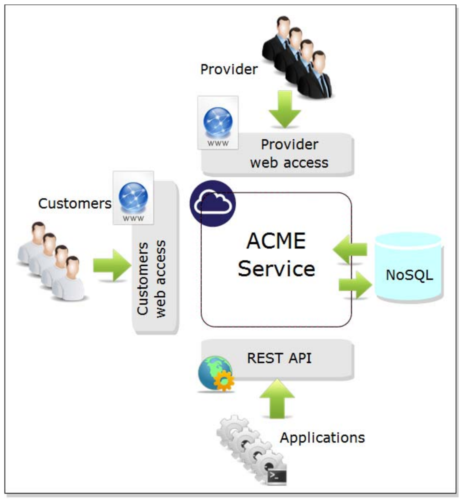

Fig. 2. ACME service architecture.

relationship holding between a customer and the bouquets he/she has subscribed.

Figure 2 shows the system components and their interactions. Providers interact with the system through a web interface to create bouquets. Customers use a different web interface to subscribe to bouquets. Finally, some parts of the information systems of the providers interact with the ACME service through a programmatic interface (e.g., SOAP or REST) to acquire information about customers' subscriptions and to trigger the modification of customers' credit based on the defined policy. Referring to the example of the gym, whenever a customer who has subscribed the gym bouquet goes to the gym, the gym information system checks through the programmatic interface if he/she has still credit and, if yes, it triggers a change in his/her credit.

III. Issues in the implementation of the ACME service with a NoSQL DB

If we try to build the ACME service by using a NoSQL DB as the main data storage, we need to face the following problems: how to flatten a complex data model on NoSQL tables and how to manage the execution of complex queries

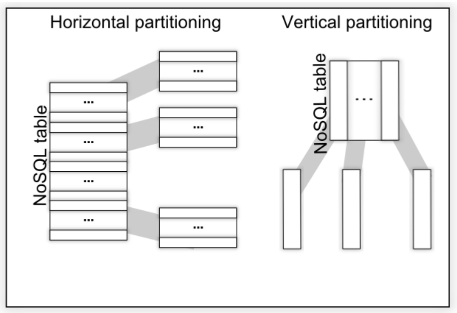

Fig. 3. Table partitioning

on the flattened structure. In the following we discuss about these two problems in detail.

A. Flattening a complex data model

A NoSQL DB is mainly organized in uncorrelated tables. Thus, a complex and intrinsically relational data model has to be flattened and partitioned in various NoSQL tables. Two different partitioning policies are possible [8], vertical partitioning, where each column of a high level relational table is stored into a separate NoSQL table and horizontal paritioning, where different tuples are stored in different NoSQL tables (see Figure 3).

The difference between the two techniques has implications on how data are stored by the underlying physical infrastructures. If NoSQL tables are stored on nodes in the same area (e.g., Amazon availability zone), a near data requires a lower access time.

In this way, the horizontal partitioning facilitates access by rows while the vertical partition is better in order to access data by columns or by column-groups. The second model is adopted by column-family NoSQL (the so called column-oriented [9]).

A different approach, typical of many NoSQL DBs [9], consists in creating objects that represent *deep copies* of the data elements in our database and store them in a key-value table, that is, in a table with two columns, one containg the key univocally representing the element and the other containing a serialization of the element itself. Referring to the ACME example, the best candidates for a deep copy are the providers entities as they are related, directly or indirectly, with all other entities of the data model (see Figure 1). From this observation, we can derive the key-value `P-table` shown in Figure 4.

The figure exhibits clearly the structure of `P-table`, consisting of a column containing the key that stores the field `id` of `P`, and a column `value` that contains a representation of `P`. This representation can be obtained by serializing the whole set of data elements correlated with the specific instance of `P` in the XML representation shown in Figure 5. As the code shows, the elements defined in Figure 1 (i.e., `P`, `C`, `B`, `R1`, `R2`, `R3`) are mapped into XML tags and the attributes (i.e., `city`, `credit`, and `policyValue`) are transformed into

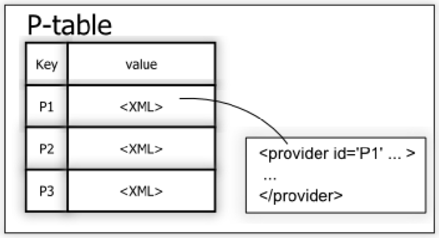

Fig. 4. ACME key-value P-table.

```
<provider id='P1' city='Milano'>
 <r1>
  <bouquet id='B1'>
   <r3>
    <customer id='C1' policy='decrease' credit='10'/>
   </r3>
  </bouquet>
  <bouquet id='B2'>
   <r3>
    <customer id='C1' policy='decrease' credit='8'/>
    <customer id='C2' policy='decrease' credit='6'/>
    <customer id='C4' policy='decrease' credit='6'/>
   </r3>
  </bouquet>
  <bouquet id='B3'>
   <r3>
    <customer id='C2' policy='increase' credit='2'/>
    <customer id='C3' policy='increase' credit='1'/>
   </r3>
  </bouquet>
 </r1>
 <r2>
  <bouquet id='B1'/>
 </r2>
</provider>
```

Fig. 5. Serialization of provider `P1`.

XML attributes.

B. Managing the execution of complex queries

The `P-table` defined in the previous section provides optimal performance when we need to access information related to a certain `P` with a certain `id`. In particular, Query 1 (expressed in a SQL-like language) is efficiently executed by selecting from `P-table` the value associated to the specified key. The result of this query is equal to the XML element shown in Figure 5.

$$SELECT\ P.value\ FROM\ P-table\ WHERE\ P.id =' P1' \quad (1)$$

Let us now consider Queries 2 and 3:

$$SELECT\ P.value\ FROM\ P-table\ WHERE\ P.city =' Milano' \quad (2)$$
$$SELECT\ P.value\ FROM\ P-table\ WHERE\ P.R1.R3.id =' C1' \quad (3)$$

The first one aims at selecting from the data set all providers located in Milano. Looking at the specific structure of our NoSQL DB, the field `city` is not a column of the `P-table` but is part of the XML encoding stored in the `value` field. This means that, to select the right data set, all `P.value` fields have to be extracted from the `P-table` and the corresponding XML documents have to be parsed.

The situation becomes even more complex when running Query 3. It returns the information about providers that have a customer with `Id` equals to `C1` (`P.R1.R3` is a set of customers related to `P` through `B`). As in the previous case, the information about all providers has to be extracted from the database and analyzed in memory to find those that are related to customer `C1` through relationships `R1` and `R3`.

IV. The Marijà framework

Currently we are developing the Marijà framework whose aim is to address the issues highlighed in the previous section. We are still far from its finalization, but we discuss here about the main ideas underlying the framework and we highlight the advantages that can derive from its usage.

Marijà works on any NoSQL DB that supports the key-value table storage approach. It is composed of two main layers, one supporting the design time activities and the other the runtime operation of the NoSQL database (see Figure 6). The design time layer supports developers in creating the logical schema of the database starting from a data model, typically provided in terms of an entity-relationship diagram, and from the definition of the most used queries for the system under development. Based on these two types of input, Marijà determines the key-value tables that are most suited to optimize the execution of the queries. As we will discuss later in more detail, it might happen that different queries require different, partially overlapping, tables to be defined. In this case, all possible tables are created and populated, and the runtime layer of Marijà is in charge of: *i)* ensuring that data updates are consistently propagated to all interested tables and *ii)* managing the execution of queries on the most suitable tables. In the following we explain in more detail the rationale for having different partially overlapping NoSQL tables and we discuss about the issues concerning data syncronization.

A. Creating partially overlapping tables

To better understand why we may need to have different co-existing and partially overlapping tables, let us consider the example of Section II. In this case, in order to optimize the execution of Query 3, Marijà creates, besides `P-table`, also `C-table` (see Figure 7), which stores data from the point of view of customers (as in the case of providers, each customer in the table is a deep copy containing information about all entities and relationships associated to the customer). Thanks to the availability of this table, instead of Query 3, we could run Query 4:

$$SELECT\ C.value\ FROM\ C-table\ WHERE\ C.id =' C1' \quad (4)$$

This would extract the information concerning customer `C1` and then we could parse, in memory, the XML serialization of `C1` to retrieve the required information about the providers. Thus, by combining the query and the XML parsing operation, we would obtain the required data with a limited effort compared to the one needed to execute Query 3 on `P-table`. In fact, while in this last case we need to extract and analyze the entire content of the database, runnig Query 4 on `C-table`

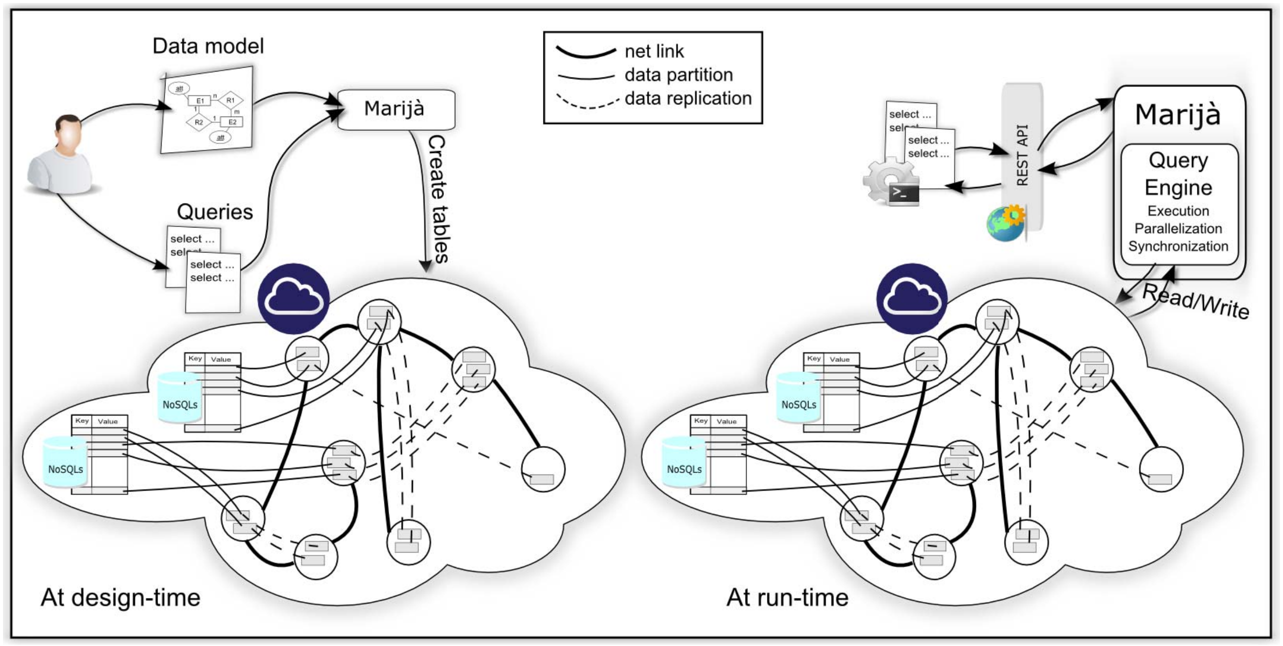

Fig. 6. Design-time and run-time workflow

limits the analysis only to the information associated to a single customer.

Thus, during the design-time phase, we need to derive, from the data model and the queries defined for a certain application, the set of NoSQL tables optimizing queries execution. These tables can be seen as materialized views on the data model and, of course, need to be coherent with it. By exploiting known results from schema transformation and schema matching literature [10], [11], we plan to build, as part of Marijà, a formal framework that automates the derivation of these views and of the corresponding NoSQL tables. This derivation requires special care since, on the one hand, a larger number of views increase the number of possible queries that the system can perform in optimized manner. On the other hand, materialized views introduce replicated data thus implying the need for synchronizing such data. This issue will be discussed in more detail in the next section.

B. Handling updates of partially replicated data

Replication is not a bad thing per se as it increases system availability, (and it is, to some extent, already supported transparently by Cloud providers for raw data), but it requires some synchronization mechanisms. If the synchronization fails, the system risks to have replicas with values that are inconsistent. For example, consider the following update query:

$$UPDATE\ P-table\ SET\ P.city =' London'\ WHERE\ P.id =' P1' \quad (5)$$

This updates `P-table`, but, to guarantee consistency, `C-table` has to be updated as well and, in particular, the following updates need to be executed:

$$\begin{aligned}&UPDATE\ C-table\ SET\ C.R3.R2.city =' London'\\ &\qquad WHERE\ C.id =' C1'\\ &UPDATE\ C-table\ SET\ C.R3.R2.city =' London'\\ &\qquad WHERE\ C.id =' C2'\\ &UPDATE\ C-table\ SET\ C.R3.R2.city =' London'\\ &\qquad WHERE\ C.id =' C3'\\ &UPDATE\ C-table\ SET\ C.R3.R2.city =' London'\\ &\qquad WHERE\ C.id =' C4'\end{aligned}$$

The lesson learned from the previous example is that we need to keep track of the relationships existing between independent NoSQL tables, based on the data model, and we need to update all related tables so that they are all consistent with each other. Since different entities are stored independently from each other, all updates can be executed in parallel. Intuitively, this can be seen by analyzing the structure of the update queries above. All of them refer to a specific provider or customer (`P1`, `C1`, `C2`, `C3`, and `C4`), that, in turn, correspond to different and independent rows of the NoSQL tables. Thus, parallel programming models like map-reduce can be easily applied in this case [12], [13]. This implies that system scalability is not necessarily hampered by synchronization.

As deeply discussed in [14], when tables are distributed on different nodes, synchronization cannot happen exactly at the same time. This means that, for a certain transitory time interval, different copies might contain different values. In this context, guaranteeing *strict consistency* requires some effort and a good distributed locking mechanism. *Eventual consistency* semantics, instead, appears to be easier to be

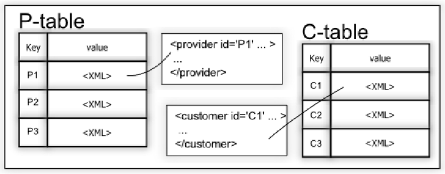

Fig. 7. P-table and C-table.

managed. The papers [8], [9] and [14] discuss in detail about this problem. We argue that the choice between the two consistency approaches has to be left to the designer of the system as it depends on the specific application being developed. Thus, Marijà has to support the designer in making the decision about the consistency semantics to enforce, by offering proper constructs. Furthermore, the cohexistence of different semantics in the same system as well as data synchronization based on the selected semantics need to be supported.

C. Applying map-reduce to perform operations

The benefits obtained from the usage of map-reduce paradigm seem natural. To show this we will briefly expose how to apply map-reduce to perform the update discussed before. As we have already seen, the update of `P-table` implies a number of updates in `C-table`. The example shows the mutual-independence among the update operations performed on the `C-table`. Every update operation is referred to a different `C-table` row and this characteristic enables the execution of several operations in parallel.

In order to apply the map-reduce methodology during the update process, we suppose to split the update task in two parts, the first one, called `updateMap`, in charge of performing the update operations on each row in parallel, and the second one, called `updateReduce`, in charge of deciding if the whole update process has been performed successfully or no. In particular, `updateReduce` collects the update results from the various `updateMap` processes and makes a decision depending on the policy it implements. This policy can be *best effort* in the case the successful update of a subset of the involved rows is considered acceptable or *guaranteed* in the case the update of all rows should be successful. In case of unsuccessful update, the `updateMap` is also in charge of triggering a roll-back procedure.

Considering our example we have `updateMap(C1,C2,C3,C4)` (with the meaning the operation will update rows `C1`, `C2`, `C3` and `C4` in C-table). The `updateMap` function launches four different tasks (in parallel) each of which aims to update each single row. To do this the `update` operation, defined below, is executed:

```
void update(Row old, Row new)
```

In our example the map function result is the creation of four tasks each of which performs the update operation:

```
update (C1Old, C1New)  (update 1)
update (C2Old, C2New)  (update 2)
update (C3Old, C3New)  (update 3)
update (C4Old, C4New)  (update 4)
```

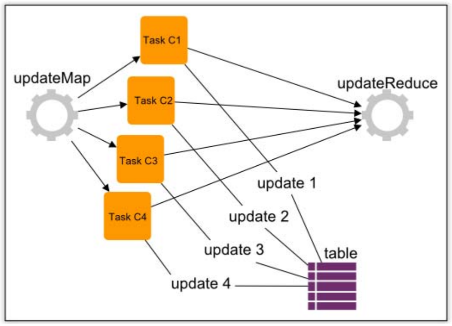

Fig. 8. Map-Reduce update

Each individual process tries to update the row and notifies the update result to the `updateReduce` process. Figure 8 summarizes the example discussed. The figure depicts the `updateMap` task that creates the update processes. When each process terminates, the result is notified to the `updateReduce` task that collects them and reacts based on its policy (e.g., fail if one update fails).

V. Related work

There are not many works addressing NoSQL issues as we plan in our research agenda. Considering the current situation, in which manufacturers offer their customized NoSQL query language, some works focus on the definition of a single language. Among these, we want to cite the UnQL. UnQL [15] is a project that comes from the collaboration of Couchbase and SQLite teams. The aim of the project is to unify the query languages of NoSQL DBs like SQL has done for RDBMS. Even though the convergence towards a unified language is a necessary step in order to improve the portability and promote the NoSQL diffusion, we argue that this is not enough to address the issues we have discussed in this paper.

In [16] the author addresses the problem of allowing access in a uniform way the data stored in NoSQL DBs and in RDBMS. This is achieved by creating an abstract layer hiding the underlying storage model. However, the approach assumes that the design of the NoSQL and RDBMS parts of the storage is given and performed outside of the proposed framework whose focus is only on the data access problem. With this respect, the approach appears to be complementary to ours as we focus on how to support developers in building a NoSQL and scalable DB starting from a complex data model.

The work proposed by Microsoft in [17] provides a theoretical framework that confirms the vision we have about NoSQLs. The paper presents a common abstract mathematical data model that binds the key-value NoSQL data model to the foreign/primary-key data model typical of RDBMS. We are confident that this mathematical theory could be useful in our project towards the realization of the Marijà framework.

VI. Conclusion and future work

NoSQL DBs represent an interesting technology in the Cloud Computing domain thanks to their scalability. As a drawback, they lack expressive power and do not support powerful data models. The goal of our research is to identify methods and mechanisms to increase the expressive power of NoSQL without impacting in a significant way on its scalability. In this position paper we have shown that exploiting tables to represent materialized views on a complex data model can be beneficial to support the optimization of query execution. This approach though has some side effects related to the need of handling data synchronization when updates are executed. We have intuitively shown that the difficulties of synchronizaton is mitigated by the possibility of parallelizing the required update operations using a map-reduce approach.

Short-term future work includes the development and the evaluation of the Marijà framework based on the lines we have presented in this paper. Longer-term work concerns the possibility to find a way that automatically discovers the best set of materialized views based on the usage of the system, without having a priori information about the queries to be executed.

Acknowledgment

This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

References

[1] N. Leavitt, "Will NoSQL Databases Live Up to Their Promise?" pp. 12–14, 2010.

[2] A. Lakshman, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, pp. 35–40, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1773922

[3] "Azure Table Storage." [Online]. Available: https://www.windowsazure.com/en-us/develop/net/how-to-guides/table-services/

[4] "SimpleDB." [Online]. Available: http://aws.amazon.com/documentation/simpledb/

[5] "DynamoDB." [Online]. Available: http://aws.amazon.com/documentation/dynamodb/

[6] F. Chang, J. Dean, S. Ghemawat, and W. Hsieh, "Bigtable: A distributed storage system for structured data," *ACM Transactions on*, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1365816

[7] "Hadoop." [Online]. Available: http://hadoop.apache.org/

[8] R. Cattell, "Scalable sql and nosql data stores," *ACM SIGMOD Record*, vol. 39, no. 4, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1978919

[9] R. Hecht and S. Jablonski, "Nosql evaluation: A use case oriented survey," in *Cloud and Service Computing (CSC), 2011 International Conference on*, dec. 2011, pp. 336 –341.

[10] Z. Bellahsene, A. Bonifati, and E. Rahm, Eds., *Schema Matching and Mapping*. Springer, 2011.

[11] R. Fagin, P. Kolaitis, and L. Popa, "Schema mapping evolution through composition and inversion," *Schema Matching and Mapping*, 2011. [Online]. Available: http://www.springerlink.com/index/M21672163264VW17.pdf

[12] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with mapreduce: a survey," *SIGMOD Rec.*, vol. 40, no. 4, pp. 11–20, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2094114.2094118

[13] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[14] S. Gilbert and N. Lynch, "Brewer ' s Conjecture and the Feasibility of Consistent , Available , Partition-Tolerant Web Services," pp. 51–59, 2005.

[15] "UnQL." [Online]. Available: http://www.unqlspec.org/display/UnQL/Home

[16] J. Roijackers, "Bridging SQL and NoSQL," Ph.D. dissertation, 2012. [Online]. Available: http://alexandria.tue.nl/extra1/afstversl/wsk-i/roijackers2012.pdf

[17] E. Meijer and G. Bierman, "A co-relational model of data for large shared data banks," *Queue*, vol. 9, no. 3, pp. 30:30–30:48, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/1952746.1961297