On bridging relational and document-centric data stores

John Roijackers and George H. L. Fletcher

Eindhoven University of Technology, The Netherlands
john@roijackers.net, g.h.l.fletcher@tue.nl

Abstract. Big Data scenarios often involve massive collections of nested data objects, typically referred to as "documents." The challenges of document management at web scale have stimulated a recent trend towards the development of document-centric "NoSQL" data stores. Many query tasks naturally involve reasoning over data residing across NoSQL and relational "SQL" databases. Having data divided over separate stores currently implies labor-intensive manual work for data consumers. In this paper, we propose a general framework to seamlessly bridge the gap between SQL and NoSQL. In our framework, documents are logically incorporated in the relational store, and querying is performed via a novel NoSQL query pattern extension to the SQL language. These patterns allow the user to describe conditions on the document-centric data, while the rest of the SQL query refers to the corresponding NoSQL data via variable bindings. We give an effective solution for translating the user query to an equivalent pure SQL query, and present optimization strategies for query processing. We have implemented a prototype of our framework using PostgreSQL and MongoDB and have performed an extensive empirical analysis. Our study shows the practical feasibility of our framework, proving the possibility of seamless coordinated query processing over relational and document-centric data stores.

1 Introduction

Nested data sets are ubiquitous in Big Data scenarios, such as business and scientific workflow management [2, 12] and web analytics [15]. The massive collections of such loosely structured data "documents" encountered in these scenarios have stimulated the recent emergence of a new breed of document-centric "NoSQL" data stores, specifically targeting the challenges of data management at web-scale [6, 17]. These solutions aim for flexible responsive management of nested documents, typically serialized in the JSON [8] data format.

While very popular and successful in many Big Data application domains, NoSQL systems will of course not displace traditional relational stores offering structured data storage and declarative querying (e.g., SQL-based access). Indeed, there is growing consensus that both NoSQL and "SQL" systems will continue to find and fulfill complementary data management needs [17]. Therefore, we can expect it to become increasingly common for users to face practical scenarios where they need to reason in a coordinated fashion across both

document-centric and relational stores. Indeed, the study presented in this paper was sparked directly by our work with a local web startup.

To our knowledge, there currently exists no unified query framework to support users in such cross-data-store reasoning. Hence, users must currently issue independent queries to the separate relational and document stores, and then manually process and combine intermediate results. Clearly, this labor-intensive and error-prone process poses a high burden to consumers of data.

Contributions. In this paper, we take the first steps towards bridging the worlds of relational and document data, with the introduction of a novel general framework, and its implementation strategies, for seamless querying over relational and document stores. Our particular contributions are as follows:

- we present a logical representation of NoSQL data in the relational model, and a query language extension for SQL, which permits seamless querying of NoSQL data in SQL;
- we develop a basic approach for processing such extended queries, and present a range of practical optimizations; and,
- we present the results of a thorough empirical analysis of our framework, demonstrating its practicality.

The solution we present here is the first to eliminate the need for ad-hoc manual intervention of the user in query (re)formulation and optimization.

Related Work. Our logical representation of documents is inspired by the so-called first-order normal form for tabular data [11] and its applications in web data integration [1, 3, 9]. Furthermore, our language extension is motivated by the successes of syntactic extensions for SQL which have been explored, e.g., for RDF graphs [7]. To our knowledge, however, our novel proposals are the first to directly address the new data and query challenges, resp., raised by NoSQL stores. The only closely related effort in this area that we are aware of is the recently proposed SOS project [4]. SOS aims at providing a uniform application programming interface to federations of heterogeneous NoSQL data stores, which is complementary to the goals of our framework.

Organization. We proceed in the paper as follows. In the next section, we introduce our theoretical framework. We then discuss query processing strategies in Sec. 3. In Sec. 4 we present the setup of our empirical study, and then in Sec. 5 we present results of our experiments. Finally, we close the paper with indications for further research in Sec. 6.

2 Theoretical Framework

In this section we present our generic framework for query processing solutions over SQL and NoSQL data stores. There are two major components of the framework. First, a logical relation available in the SQL database representing

(arbitrary) NoSQL data is introduced in Sec. 2.1. Second, to seamlessly and transparently query the NoSQL data from SQL, we present an SQL query language extension with a JSON-like syntax in Sec. 2.2. We conclude the section with an overview of the entire theoretical framework.

2.1 Logical representation of NoSQL data in the relational model

In this paper, we model NoSQL *documents* as finite sets of key-value pairs, where values themselves can be finite sets of key-value pairs, and where keys and atomic values are elements of some infinite universe (e.g., Unicode strings). For the sake of simplicity, and without any loss of generality, we omit other features available in the JSON data model, such as ordered list values.

Disregarding implementation details for the moment, we assume that in the relational store there is a relation $F(id, key, value)$ available containing records representing the NoSQL data. This is a potentially virtual relation, i.e., F is implemented as a non-materialized view on the NoSQL data. The basic idea is that each document d in the NoSQL data set is assigned a unique identifier i_d. Then, for each key-value pair $k : v$ of d, a triple (i_d, k, v) is added to F. If v is a nested value, then it is assigned a unique identifier, and the process recurs on the elements of v.

We illustrate F via an example. Suppose we have product data in the document store, with nested supplier data to identify the supplier and to keep track of current stock. Then an example snippet of F, containing three product documents i_1, i_2, and i_3, is as follows:

$$\begin{aligned} F = \{&(i_1, \text{name}, \textit{monitor}), (i_1, \text{category}, 7), (i_1, \text{color}, \textit{black}), (i_1, \text{supplier}, i_4), \\ &(i_2, \text{name}, \textit{mouse}), (i_2, \text{category}, 37), (i_2, \text{color}, \textit{pink}), (i_2, \text{supplier}, i_5), \\ &(i_3, \text{name}, \textit{keyboard}), (i_3, \text{category}, 37), (i_3, \text{supplier}, i_6), \\ &(i_4, \text{id}, 1), (i_4, \text{stock}, 1), (i_5, \text{id}, 3), (i_5, \text{stock}, 5), (i_6, \text{id}, 1)\}. \end{aligned}$$

For example, i_1 corresponds to the document

$$\{\text{name} : \textit{monitor}, \text{category} : 7, \text{color} : \textit{black}, \text{supplier} : \{\text{id} : 1, \text{stock} : 1\}\}.$$

Here, we see that the "supplier" key of i_1 has a nested value, given the unique identifier i_4. Clearly, the F representation of a document store is well-defined. Due to space limitations we omit a formal definition and refer the reader to the full version of this paper for further details [16].

The F representation has two important advantages. Firstly, it is a basic and extremely flexible way to describe data which can be used to denote any possible type of NoSQL data, including documents with missing and/or set-valued keys. Secondly, F has a fixed schema and thus the (potentially schema-less) non-relational data can be accessed via a standard table in the SQL database.

The relation F can be used in queries just like any other relation in the relational database. This means we can join F to other relations to combine SQL and NoSQL data to construct a single query result. Important to note

however, is that the records in F are retrieved from the external NoSQL data source on the fly. To return the records of F, the SQL database has to retrieve the external NoSQL data in triple format.

This implies that the retrieved data has to be communicated to the relational database, which is dependent on the exact implementation of F and the method used to retrieve data from the NoSQL source. We assume that information regarding to which NoSQL source F should connect and what data to retrieve is given. In practice, the implementation of F will of course be parameterized, to specify which NoSQL database should be used. For readability however, we ignore such parameters. In the remainder of this paper we will thus simply use F as the relation of triples that represents the NoSQL data. We discuss implementation details further in Sec. 4, below.

2.2 Declarative querying over SQL and NoSQL stores

A series of self joins of F on the *id* field allows reconstruction of the NoSQL data. However, in cases where many self joins are needed (e.g., to retrieve data elements in deeply nested documents) this is a tedious error-prone task. To facilitate querying the NoSQL data more conveniently and avoid the task of manually adding join conditions, we introduce an SQL extension which we call a *NoSQL query pattern* (NQP).

An NQP is based on the concept of variable bindings as used in standard conjunctive query patterns which are at the core of well-known languages such as Datalog, SQL, SPARQL, and XPath [1]. Consider a graph pattern where triples can be nested, and triple elements can be either variables or atomic values. All triples on the same nesting level describe the same triple subject and therefore the identifier of this element can be excluded from the query. The result is a JSON-like nested set of key-value pairs where some elements may be variables.

For example, suppose that on the SQL side we have a table of product suppliers, with schema *Supplier*(*id*, *name*, *region*), and on the NoSQL side the previously mentioned product data (this is an actual real-world situation which we encountered in our collaboration with a local web startup). The query given in Listing 1.1 retrieves product name and supplier region information for products in category 37 having a minimum stock of 2.

```
1  SELECT
2    p.n, s.region
3  FROM
4    NoSQL( name: ?n, category: 37, color: ?c,
5           supplier: ( id: ?i, stock: ?s ) ) AS p,
6    Supplier AS s
7  WHERE
8    p.i = s.id AND p.s >= 2
```

Listing 1.1: Example SQL query with an NQP

Besides illustrating the NQP syntax, the example also demonstrates how NQP's are included in an SQL query. This method allows easy isolation of the

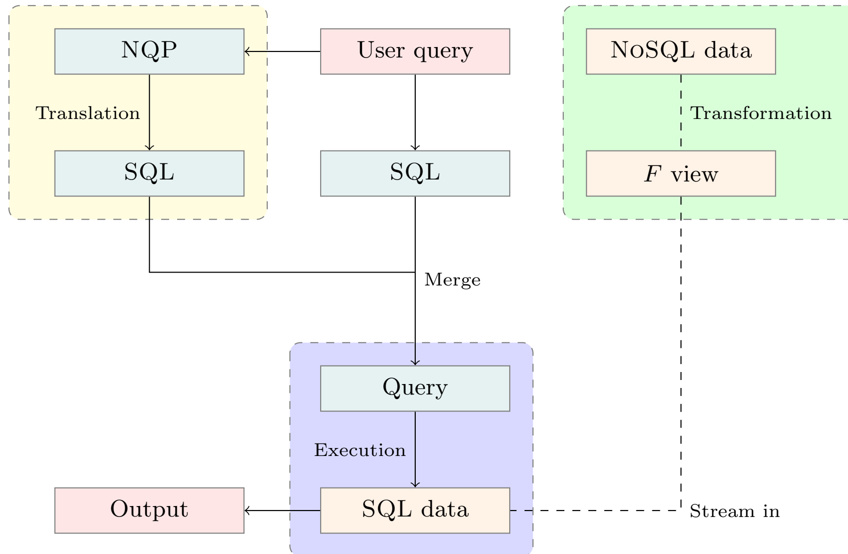

Fig. 1: Architectural workflow illustrating the life of a SQL+NoSQL query.

NQP and references to the variable bindings in the rest of the query. In the SQL part of the query the variables in the NQP can be treated as relation attributes from the virtual NoSQL relation. In Sec. 3 we describe a method to automatically translate an arbitrary NQP to a pure SQL equivalent.

The NQP extension provides a convenient way to describe which NoSQL data is required and how it should be reconstructed on the SQL side, without exposing to the user the underlying representation and query processing. As far as the user is concerned, the NoSQL data is contained in a single native relation.

In addition to user transparency and simplicity, an additional major advantage of our approach is its independence from the underlying NoSQL database. The use of another document store implies a new implementation of the F relation. However, SQL+NQP queries, and hence the applications in which they are embedded, are not affected by the change of the underlying database.

Architectural Overview. To summarize, we give an overview of our theoretical framework in Fig. 1. Documents, stored in a NoSQL database, are made available in the relational database via a logical relation F. This data is streamed in on demand, so the NoSQL database is still used as the primary store for the document data. A user can then issue an SQL query containing NQP's, which describes constraints on the desired NoSQL data and their combination with the SQL data. The query is then automatically translated behind the scenes to a pure SQL query and processed by the relational database.

3 Query Processing Strategies

In this section we introduce implementation strategies for putting our theoretical framework of the previous section into practice. The NQP extension, based on a virtual relational representation of document data, must be automatically translated to a pure SQL equivalent to be executed by an SQL database. In Sec. 3.1 we provide a baseline approach to this translation. Sec. 3.2 focuses on optimization techniques to accelerate the performance of this base translation.

3.1 Base Translation

The nested key-value pairs in an NQP can be recursively numbered as follows:

$$\text{NoSQL}(k_1 : v_1, k_2 : v_2, \ldots, k_r : (k_{r,1} : v_{r,1}, k_{r,2} : v_{r,2}, \ldots, k_{r,m} : v_{r,m}) \ldots, k_n : v_n)$$

The basic idea behind a translation of an NQP is that each key-value pair $t = k_t : v_t$ is represented in SQL using its own copy F_t of the logical relation F. For these copies we introduce a shorthand notation to avoid naming conflicts and to apply selection conditions, specified as follows:

F_t	**Constant** k_t	**Variable** k_t
Constant v_t	$\rho_{F_t(\mathrm{i}_t,\mathrm{k}_t,\mathrm{v}_t)}\left(\sigma_{\mathrm{key}=k_t \wedge \mathrm{value}=v_t}\left(F\right)\right)$	$\rho_{F_t(\mathrm{i}_t,\mathrm{k}_t,\mathrm{v}_t)}\left(\sigma_{\mathrm{value}=v_t}\left(F\right)\right)$
Variable v_t	$\rho_{F_t(\mathrm{i}_t,\mathrm{k}_t,\mathrm{v}_t)}\left(\sigma_{\mathrm{key}=k_t}\left(F\right)\right)$	$\rho_{F_t(\mathrm{i}_t,\mathrm{k}_t,\mathrm{v}_t)}\left(F\right)$
Nested v_t	$\rho_{F_t(\mathrm{i}_t,\mathrm{k}_t,\mathrm{v}_t)}\left(\sigma_{\mathrm{key}=k_t}\left(F\right)\right)$	$\rho_{F_t(\mathrm{i}_t,\mathrm{k}_t,\mathrm{v}_t)}\left(F\right)$

Each copy of F is subscripted with t, just like each of its attributes. Depending on the type of both the key and value of the pair, we can add a selection operator to exclude unnecessary triples from F_t on the SQL side. When neither the key nor the value is a constant value, F_t is only a renamed version of F. This is inefficient for query processing, because F_t then contains all triples in F, and thus potentially the entire NoSQL dataset.

The next translation step is reconstruction of the NoSQL data by correctly joining the F_t copies. For key-value pairs at the same nesting level in the NQP this means that the *id* attributes should have the same value, since the F_t relations for an NoSQL object have been given identical *id* values. For a nested NQP the translation is applied recursively. The triple relations are created such that nested data is connected via a *value* attribute that is equal to the *id* value of the nested triples. Combining the correct triples is therefore only a matter of inserting the correct join condition for the nested set of F copies.

Using this method, the NQP of the example query from Listing 1.1 is translated to the following series of joins:

$$F_{\mathrm{name}} \bowtie_{\mathrm{i_{name}}=\mathrm{i_{category}}} F_{\mathrm{category}} \bowtie_{\mathrm{i_{category}}=\mathrm{i_{supplier}}} F_{\mathrm{supplier}} \bowtie_{\mathrm{v_{supplier}}=\mathrm{i_{supplier.id}}} \left(F_{\mathrm{supplier.id}} \bowtie_{\mathrm{i_{supplier.id}}=\mathrm{i_{supplier.stock}}} F_{\mathrm{supplier.stock}}\right) \bowtie_{\mathrm{i_{supplier}}=\mathrm{i_{color}}} F_{\mathrm{color}}.$$

Note that since the nested relations $F_{r,t}$ have equal *id* values, a single join constraint is sufficient. Because of the introduced shorthand notation, this query

applies the correct selection criteria to each relation F_t, renames the attributes and combines the corresponding triples.

In the rest of the query outside of the NQP, references to variables in the NQP occur. This means that we must keep track of the variables used in the NQP and use this information to adjust the SQL part of the query accordingly. To achieve this we introduce the function *insert*(v, a) defined as:

$$insert(v,a) = \begin{cases} V_v = \{a\}, & \text{if } V_v \text{ does not exist} \\ V_v = V_v \cup \{a\}, & \text{otherwise.} \end{cases}$$

We call this function for each variable in the NQP. Now, letting $\mathcal{V}$ be the set of variables encountered, we can replace the references to these variables in the SQL part of the query by an attribute from a triple relation copy as follows: $\forall_{v \in \mathcal{V}} Q\,[\text{r}.v := e_v]$, for an arbitrary $e_v \in V_v$.

Now the NQP has been translated, all relations are joined correctly, and variable references in the SQL part of the query have been replaced by appropriate relation attributes. Finally, we add an additional selection condition to the query which ensures that if the same variable is used multiple times in the NQP, it has an equal value. Formally and without minimizing the number of equalities this means: $\bigwedge_{v \in \mathcal{V}} \bigwedge_{i,j \in V_v} i = j$.

3.2 Optimizations

We next briefly introduce optimization strategies, to improve upon the naive base translation of the previous section. Further details can be found in [16].

(1) Data filtering. Although the naive approach from Sec. 3.1 results in a correct SQL translation, this method has a significant disadvantage. For each copy of F_t the entire NoSQL dataset must be transformed to triples and shipped to the SQL database. A straightforward way to improve performance is to filter NoSQL data that is not required to answer the query.

Firstly, we introduce a parameter c for the relations F_t. This parameter is used to describe a conjunction of selection conditions. These conditions are pushed down to the NoSQL database to restrict the number of documents that have to be sent to the SQL database. Also, in addition to conventional relational algebra conditions, more advanced selection conditions, like the existence of a field in the NoSQL database, can be pushed down:

$$c = \begin{cases} k_t = v_t, & \text{if both } k_t \text{ and } v_t \text{ have a constant value} \\ exists(k_t), & \text{if only } k_t \text{ has a constant value} \\ true, & \text{otherwise.} \end{cases}$$

Moreover, while translating the NQP to SQL, we can collect all selections and combine them in a single condition that describes the entire constraint on the NoSQL data we can derive from the NQP. Because this is a conjunction of criteria, more documents are filtered from each F_t copy.

Additionally, we derive new NoSQL selection conditions from the SQL part of the query. Moreover, we can also apply transitivity on the available conditions to put even more selective constraints on the NoSQL data.

For our example this means that the SQL selection `p.s >= 2` can be included in the final selection condition used to filter the NoSQL documents: $c = \mathit{exists}(\text{name}) \wedge \text{category} = 37 \wedge \mathit{exists}(\text{color}) \wedge \mathit{exists}(\text{supplier}) \wedge \mathit{exists}(\text{supplier.id}) \wedge \text{supplier.stock} \geq 2$. Note that we adopt a non-null semantics for simplicity.

This *document filter pushdown* reduces the number of NoSQL documents. The matching NoSQL objects themselves, however, are completely transformed to triples before they are shipped to the SQL side. We can further reduce the size of the NoSQL data by excluding triples representing attributes that are not used elsewhere in the SQL query. Hence, we introduce a parameter p for the relations F_t, to describe an *attribute filter pushdown* condition on the NoSQL data to eliminate shipping of these unused attributes.

To determine if a given key-value pair t must be shipped to the SQL database, we recursively determine if t contains a variable used in the SQL part of the query, or, in case v_t is nested, if there exists a nested key-value pair under t which should be in the translation. Note that the F_t copy representing t is no longer required in the query. In addition to decreasing the amount of data that has to be shipped from NoSQL to SQL, this optimization will lead to a final query which also contains fewer relations and thus requires fewer joins to reconstruct the NoSQL data in the relational database.

For the example query this means we can exclude F_{color} and F_{category}, since these are not used except as selection conditions on the NoSQL data. The reduction of the number of triples is achieved by the following projection argument: $p = \{\text{name}, \text{supplier}, \text{supplier.id}, \text{supplier.stock}\}$. Note that F_{supplier} is still necessary to retrieve the supplier and its current stock count.

(2) Temporary table. Each relation copy F_t requires a new connection to the external NoSQL database. To avoid this overhead we can create a temporary relation T equal to F prior to a query execution. We then change the translation such that instead of F the temporary relation T is used. As a result all triples are communicated to the SQL database only once. Document and attribute filter pushdown can now be applied to this temporary table.

(3) Tuple reconstruction. Triples are joined based on shared *id* values, matching a higher level *value* in case of nested data. This means that for nested data it is not possible for an arbitrary pair of triples in F to determine whether or not they belong to the same NoSQL data object. This information could be used when joining the triples and thereby speed up the NoSQL data reconstruction in the relational database.

To achieve this, we add an additional attribute *nosql* to F that indicates the NoSQL document to which the record belongs. Triples originating from the same document get equal *nosql* values, using the native NoSQL document identifier. Only triples from the same NoSQL document are combined on the SQL side, so this extra information can speed up the join.

At the creation of the temporary table we use the *nosql* attribute to include additional selection conditions. We create a hash index on the *nosql* attribute and include an equality selection condition on this attribute for each F_t used in the SQL query. This way, the relational database has additional knowledge about the triple structure and is able to combine triples that belong to the same NoSQL document, regardless of the nesting level.

4 Experimental Setup

With the developed theoretical framework and the automatic query translation and its optimizations, we implemented a prototype version of the hybrid database to provide a proof of concept that the theoretical framework is practically feasible. In this section, we first elaborate on the specific SQL and NoSQL databases we used and how the virtual relation F is implemented. Following this, we then discuss the experiment setup, including the data and benchmark queries used in our study. Interested readers will find deeper details on all aspects of the experimental setup in the full version of this paper [16].

Environment. In our empirical study we use MongoDB 2.0.2, an open-source and widely used document-oriented NoSQL data store.[1] For the relational database we use PostgreSQL 9.1.2.[2], an open source, industrial strength, widely used system, which offers great extensibility. The latter is particularly useful for the implementation of F.

The logical F relation is implemented as a foreign table, using a foreign data wrapper (FDW) as described in the SQL/MED extension of the SQL standard [14]. From a user perspective a foreign table is a read-only relation similar to any other relation. The technical difference is that the data in a foreign table is retrieved from an external source at query time. We use the Multicorn 0.0.9 FDW to implement the foreign table.[3] This is an existing PostgreSQL extension that uses the output of a Python script as the external data source. In our case the script retrieves the data from MongoDB and performs necessary transformations on the documents.

For the experiment we use a machine with 2 quad core Intel Xeon E5640 processors, 36 GB of memory, and 4 10 krpm SAS hard disks in RAID 5 running on Debian wheezy/sid for the PostgreSQL and MongoDB database. Due to practical constraints a virtual machine on the same machine is responsible for running the experiment. The virtual machine runs on the same OS, uses 2 processor cores and has 2 GB of memory available. The experiment process sends a query to be executed to the main machine, retrieves the result, and performs time measurements.

[1] http://www.mongodb.org/
[2] http://www.postgresql.org/
[3] http://multicorn.org/

Experiments. The experiment is twofold. We start with a small *Experiment 1* in which we focus on the feasibility of the prototype implementation and the effect of using a temporary relation prior to the actual query execution. We compare implementations $\mathcal{I}_a$ and $\mathcal{I}_b$. Implementation $\mathcal{I}_a$ is the base translation with only the data filtering optimization included. In $\mathcal{I}_b$ the temporary table is also implemented. For both implementations the number of documents returned is limited to 100.

In *Experiment 2* we compare two implementations to see what the impact of the tuple reconstruction optimization is in a larger scale setting. Here we have implementation $\mathcal{I}_c$, similar to $\mathcal{I}_b$, and implementation $\mathcal{I}_d$ with the tuple reconstruction included. For this experiment we increase the limit on the number of MongoDB documents returned to 25 000.

For both experiments the performance of the implementations is measured in terms of query execution time. Query translation, logical table generation, indexing, and actual query execution are measured separately.

Data. We use multiple datasets for a detailed comparison. The first dataset is constructed using a set of real life product data taken from a company's production environment. This NoSQL data is similar to our running example above, and can be joined to two relations in the SQL database.

Using this product set as a basis, we create different datasets by varying three parameters. Firstly, the number of products in the NoSQL dataset is either low or high. In our case this means 100 000 or 400 000 documents respectively. Also, we vary the SQL data size between 1000 and 10 000 records. And finally we cover different join probability between SQL and NoSQL data by using 0.05, 0.20, and 1.00. We use all possible combinations of these variations and thus create 12 datasets based on the product data. Each of these datasets is named $\mathcal{S}_{n,s,j}$, where n, s, and j describe the NoSQL size, SQL size, and join probability respectively.

To see the impact of a totally different dataset, we also use a *Twitter* dataset. For this dataset we collected a coherent set of 500 000 tweets about a single subject posted between March 5 and March 11 of 2012 using the *Twitter Search API*. These tweets have information about users and languages included, which we store in an SQL database to create a possibility to join the non-relational *Twitter* data to SQL data. We use $\mathcal{S}_t$ to denote this dataset.

Queries. The manner in which a query uses SQL and NoSQL data influences the query execution time. There are different ways to combine data from both stores regarding the data source on which the query mainly bases its selections. We use the term *flow class* to indicate how the query is constructed. We consider four flow classes: $\mathcal{F}_{\mathrm{i}}$ (queries only selecting NoSQL data); $\mathcal{F}_{\mathrm{ii}}$ (queries selecting SQL data on the basis of NoSQL data); $\mathcal{F}_{\mathrm{iii}}$ (queries selecting NoSQL data on the basis of SQL data); and, $\mathcal{F}_{\mathrm{iv}}$ (queries selecting SQL data on the basis of NoSQL data selected using SQL data). Other flow classes can be studied, but these cover the basic ways to combine data.

Besides the flow class of a query, query execution can also depend on how the NoSQL data is used in the SQL query. We distinguish two query properties: (a) uses all mentioned NoSQL keys in the SQL part of the query; and, (b) all mentioned NoSQL keys exist in the data for all documents. We create different *query types* based on the possible combinations of these two properties, namely: $\mathcal{Q}_1$ $((a) \wedge (b))$; $\mathcal{Q}_2$ $((a) \wedge \neg(b))$; $\mathcal{Q}_3$ $(\neg(a) \wedge (b))$; and, $\mathcal{Q}_4$ $(\neg(a) \wedge \neg(b))$. The example query of Listing 1.1 above falls into flow class $\mathcal{F}_{\mathrm{ii}}$ and query type $\mathcal{Q}_4$.

For each dataset we construct a set of query templates, for each combination of a flow class and a query type. For each dataset we thus have a total of 16 query templates. From each query template 16 concrete queries are created by filling in random values for placeholders included in the template. In our experiments, we evaluate and measure each concrete query 5 times. For the empirical analysis we drop the highest and lowest value from the 5 repetitions and report the average; we also drop the 3 highest and lowest averages per template.

5 Empirical Analysis

In this section we discuss the results of Experiments 1 and 2. We start with a presentation of the results of the first experiment, where we analyze the effect of using a temporary relation. This is followed by a discussion of the second experiment, where the NoSQL data reconstruction optimization is used. Finally, we discuss the main conclusions which can be drawn from these investigations.

Results of Experiment 1. In Experiment 1 we look at the effect of creating a single temporary relation containing all required triples prior to the actual query execution on the SQL side in a small scale experiment. In Table 1 the average result per query flow class and query type are presented. From these results, it is clear that the use of a temporary table is a significant improvement, except for $\mathcal{Q}_3$ within $\mathcal{F}_{\mathrm{i}}$. For this exception however, the initial result for $\mathcal{I}_a$ was already low and the result for $\mathcal{I}_b$ still is the best for all flow class and query type combinations.

In Table 2 we see that the *Twitter* dataset in particular profits from this optimization. For *Twitter*, the average query time was 21.7521 s compared to 0.4785 s with the use of a temporary relation. For the average product dataset, on the other hand, the average result dropped from 9.8303 s to 0.5789 s.

Results of Experiment 2. The larger-scale second experiment focuses on the effects of optimizing the NoSQL data reconstruction in the relational database by including additional information with each triple. Similar to the analysis of Experiment 1, Table 3 provides an overview per query flow class and query type.

For $\mathcal{F}_{\mathrm{i}}$ and $\mathcal{F}_{\mathrm{ii}}$ the added SQL attribute creates a small overhead. As a result, the tuple reconstruction is not an improvement and hence we omit their details in the table. For $\mathcal{F}_{\mathrm{i}}$ and $\mathcal{F}_{\mathrm{ii}}$ we observed that the average time increased from 11.5800 s and 12.5575 s to 12.4004 s and 13.1219 s respectively.

The other flow classes, $\mathcal{F}_{\mathrm{iii}}$ and $\mathcal{F}_{\mathrm{iv}}$, indicate that optimizing the tuple reconstruction can have a positive effect on the performance of the prototype. Both

Table 1: Comparison of $\mathcal{I}_a$ and $\mathcal{I}_b$ performance in seconds per flow class.

(a) Flow class $\mathcal{F}_{\text{i}}$

	$\mathcal{I}_a$	$\mathcal{I}_b$	$\mathcal{I}_b$ / $\mathcal{I}_a$
$\mathcal{Q}_1$	4.1123	0.4766	0.116
$\mathcal{Q}_2$	22.0757	0.6694	0.030
$\mathcal{Q}_3$	0.2467	0.4021	1.630
$\mathcal{Q}_4$	4.6585	0.4469	0.096

(b) Flow class $\mathcal{F}_{\text{ii}}$

	$\mathcal{I}_a$	$\mathcal{I}_b$	$\mathcal{I}_b$ / $\mathcal{I}_a$
$\mathcal{Q}_1$	4.8076	0.5232	0.109
$\mathcal{Q}_2$	14.7703	0.7101	0.048
$\mathcal{Q}_3$	3.1227	0.4493	0.144
$\mathcal{Q}_4$	6.7764	0.4309	0.064

(c) Flow class $\mathcal{F}_{\text{iii}}$

	$\mathcal{I}_a$	$\mathcal{I}_b$	$\mathcal{I}_b$ / $\mathcal{I}_a$
$\mathcal{Q}_1$	10.9531	0.6207	0.057
$\mathcal{Q}_2$	38.5549	0.8354	0.022
$\mathcal{Q}_3$	5.7535	0.4733	0.082
$\mathcal{Q}_4$	18.7430	0.5599	0.030

(d) Flow class $\mathcal{F}_{\text{iv}}$

	$\mathcal{I}_a$	$\mathcal{I}_b$	$\mathcal{I}_b$ / $\mathcal{I}_a$
$\mathcal{Q}_1$	8.6195	0.6662	0.077
$\mathcal{Q}_2$	11.7808	0.8811	0.075
$\mathcal{Q}_3$	6.7123	0.4301	0.064
$\mathcal{Q}_4$	10.2702	0.5645	0.055

Table 2: Comparison of $\mathcal{I}_a$ and $\mathcal{I}_b$ performance in seconds per dataset.

	$\mathcal{F}_{\text{i}}$			$\mathcal{F}_{\text{ii}}$			$\mathcal{F}_{\text{iii}}$			$\mathcal{F}_{\text{iv}}$		
	$\mathcal{I}_a$	$\mathcal{I}_b$	$\mathcal{I}_b$ / $\mathcal{I}_a$	$\mathcal{I}_a$	$\mathcal{I}_b$	$\mathcal{I}_b$ / $\mathcal{I}_a$	$\mathcal{I}_a$	$\mathcal{I}_b$	$\mathcal{I}_b$ / $\mathcal{I}_a$	$\mathcal{I}_a$	$\mathcal{I}_b$	$\mathcal{I}_b$ / $\mathcal{I}_a$
$\mathcal{S}_{l,l,l}$	5.5666	0.5054	0.091	3.3699	0.5207	0.154	26.5421	0.6488	0.024	7.7278	0.6041	0.078
$\mathcal{S}_{l,l,m}$	5.6544	0.4989	0.088	2.6993	0.5134	0.190	12.9048	0.6137	0.048	9.4213	0.6354	0.067
$\mathcal{S}_{l,l,h}$	5.6304	0.5043	0.090	10.2625	0.5691	0.055	10.6876	0.6416	0.060	15.1030	0.6971	0.046
$\mathcal{S}_{l,h,l}$	5.8085	0.4940	0.085	1.5479	0.5193	0.335	31.5640	0.6483	0.021	6.8629	0.6285	0.092
$\mathcal{S}_{l,h,m}$	5.8462	0.4967	0.085	4.3644	0.5171	0.118	12.4716	0.6125	0.049	7.5289	0.6451	0.086
$\mathcal{S}_{l,h,h}$	5.9960	0.5000	0.083	10.3962	0.5720	0.055	11.0061	0.6555	0.060	16.1746	0.7054	0.044
$\mathcal{S}_{h,l,l}$	6.0348	0.4958	0.082	5.3076	0.5169	0.097	33.0351	0.6581	0.020	7.9743	0.5678	0.071
$\mathcal{S}_{h,l,m}$	5.9849	0.4984	0.083	1.6218	0.5066	0.312	13.2531	0.6295	0.047	1.6119	0.5764	0.358
$\mathcal{S}_{h,l,h}$	5.5507	0.5038	0.091	9.9075	0.5476	0.055	10.8158	0.6367	0.059	14.8312	0.7086	0.048
$\mathcal{S}_{h,h,l}$	5.9669	0.5192	0.087	1.5943	0.5079	0.319	31.0614	0.6595	0.021	6.1285	0.6327	0.103
$\mathcal{S}_{h,h,m}$	5.4860	0.4984	0.091	8.1835	0.5415	0.066	13.0176	0.6209	0.048	8.1223	0.6426	0.079
$\mathcal{S}_{h,h,h}$	5.6549	0.4814	0.085	10.1422	0.5555	0.055	10.6277	0.6359	0.060	14.8031	0.7003	0.047
$\mathcal{S}_t$	31.8727	0.4870	0.015	26.4032	0.4813	0.018	23.5278	0.4290	0.018	5.2046	0.5169	0.099
Avg	7.7733	0.4987	0.064	7.3693	0.5284	0.072	18.5011	0.6223	0.034	9.3457	0.6355	0.068

these flow classes focus on selecting SQL data before joining NoSQL data. Especially $\mathcal{Q}_2$ within $\mathcal{F}_{\text{iii}}$ shows that the average query time can be significantly reduced when the NoSQL data reconstruction is optimized.

When comparing the different datasets in Table 4 we again exclude $\mathcal{F}_{\text{i}}$ and $\mathcal{F}_{\text{ii}}$ since the optimization does not have an effect for these flow classes. For the other flow classes average results per dataset are presented. Again we see that the optimization typically results in a significant performance increase.

Discussion. After analyzing both experiments separately, we can draw some broad conclusions regarding the framework and optimizations. Firstly, Experiment 1 shows that a working prototype can be constructed based on the proposed framework. Furthermore, a temporary relation significantly improves the

Table 3: Comparison of $\mathcal{I}_c$ and $\mathcal{I}_d$ performance in seconds per flow class.

(a) Flow class $\mathcal{F}_{\text{iii}}$

	$\mathcal{I}_c$	$\mathcal{I}_d$	$\mathcal{I}_d$ / $\mathcal{I}_c$
$\mathcal{Q}_1$	13.5964	11.3169	0.832
$\mathcal{Q}_2$	34.1419	7.7161	0.226
$\mathcal{Q}_3$	9.2465	9.6599	1.045
$\mathcal{Q}_4$	7.0996	5.7576	0.811

(b) Flow class $\mathcal{F}_{\text{iv}}$

	$\mathcal{I}_c$	$\mathcal{I}_d$	$\mathcal{I}_d$ / $\mathcal{I}_c$
$\mathcal{Q}_1$	15.5227	11.8015	0.760
$\mathcal{Q}_2$	12.8297	8.9534	0.698
$\mathcal{Q}_3$	11.8088	10.2127	0.865
$\mathcal{Q}_4$	8.2041	6.1557	0.750

Table 4: Comparison of $\mathcal{I}_c$ and $\mathcal{I}_d$ performance in seconds per dataset.

	$\mathcal{F}_{\text{iii}}$			$\mathcal{F}_{\text{iv}}$		
	$\mathcal{I}_c$	$\mathcal{I}_d$	$\mathcal{I}_d$ / $\mathcal{I}_c$	$\mathcal{I}_c$	$\mathcal{I}_d$	$\mathcal{I}_d$ / $\mathcal{I}_c$
$\mathcal{S}_{l,l,l}$	8.0736	1.3646	0.169	8.8565	1.7372	0.196
$\mathcal{S}_{l,l,m}$	9.6559	4.7486	0.492	6.7734	5.4149	0.799
$\mathcal{S}_{l,l,h}$	11.4418	11.5517	1.010	12.1739	13.2877	1.091
$\mathcal{S}_{l,h,l}$	9.1960	1.2582	0.137	11.1787	1.6435	0.147
$\mathcal{S}_{l,h,m}$	9.6356	4.4673	0.464	12.5762	5.2294	0.416
$\mathcal{S}_{l,h,h}$	11.3714	13.1159	1.153	12.8711	13.2781	1.032
$\mathcal{S}_{h,l,l}$	7.3243	4.2647	0.582	7.7039	3.2565	0.423
$\mathcal{S}_{h,l,m}$	11.3422	12.2677	1.082	13.8836	13.1399	0.946
$\mathcal{S}_{h,l,h}$	70.5085	20.8496	0.296	23.5845	22.5808	0.957
$\mathcal{S}_{h,h,l}$	11.1132	4.0471	0.364	6.6920	4.3302	0.647
$\mathcal{S}_{h,h,m}$	11.7446	12.8316	1.093	12.5065	13.6754	1.093
$\mathcal{S}_{h,h,h}$	35.7321	20.1385	0.564	22.7828	22.1383	0.972
$\mathcal{S}_t$	1.1351	1.0585	0.933	5.6044	0.9391	0.168
Avg	16.0211	8.6126	0.538	12.0913	9.2808	0.768

performance by an order of magnitude. The second experiment, conducted in a larger-scale setting shows that the NoSQL data reconstruction strategy is indeed a successful optimization for many query classes. The final prototype, $\mathcal{I}_d$, while successful as a proof-of-concept, leaves space open for future improvements. In general, we conclude from our study the practical feasibility of the proposed query processing framework.

6 Conclusions

In this paper we have presented the first generic extensible framework for coordinated querying across SQL and NoSQL stores which eliminates the need for ad-hoc manual intervention of the user in query (re)formulation and optimization. An extensive empirical study demonstrated practical feasibility of the framework and the proposed implementation strategies.

The groundwork laid here opens many interesting avenues for further research. We close by listing a few promising directions. (1) We have just scratched

the surface of implementation and optimization strategies. We give two suggestions for future work here. (a) We can study adaptations and extensions of indexing and caching mechanisms developed for RDF, a triple-based data model, and XML to more scalable implementations of F [13, 18]. (b) Individual queries are often part of a longer-running collection. It would be certainly worthwhile to investigate strategies for multi-query optimization with respect to a dynamic query workload. (2) There is recent work in the community towards standardization of document query languages and their semantics [5, 10, 19]. An important interesting topic for further investigation is to coordinate our results with these emerging efforts (e.g., studying appropriate extensions or restrictions to NQP's).

Acknowledgments. We thank T. Calders and A. Serebrenik for their feedback.

References

1. S. Abiteboul et al. *Web data management.* Cambridge University Press, 2011.
2. U. Acar et al. A graph model of data and workflow provenance. In *Proc. TAPP*, San Jose, California, 2010.
3. R. Agrawal, A. Somani, and Y. Xu. Storage and querying of e-commerce data. In *Proc. VLDB*, pages 149–158, Rome, 2001.
4. P. Atzeni et al. Uniform access to non-relational database systems: The SOS platform. In *Proc. CAiSE*, pages 160–174, Gdansk, Poland, 2012.
5. V. Benzaken, G. Castagna, K. Nguyen, and J. Siméon. Static and dynamic semantics of NoSQL languages. In *Proc. ACM POPL*, pages 101–114, Rome, 2013.
6. R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, December 2010.
7. E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *Proc. VLDB*, pages 1216–1227, Trondheim, Norway, 2005.
8. D. Crockford. The application/json media type for javascript object notation (JSON). http://www.ietf.org/rfc/rfc4627.txt, 2006.
9. G. H. L. Fletcher and C. M. Wyss. Towards a general framework for effective solutions to the data mapping problem. *J. Data Sem.*, 14:37–73, 2009.
10. JSONiq. http://jsoniq.org .
11. W. Litwin, M. Ketabchi, and R. Krishnamurthy. First order normal form for relational databases and multidatabases. *SIGMOD Record*, 20(4):74–76, 1991.
12. B. Ludäscher, M. Weske, T. M. McPhillips, and S. Bowers. Scientific workflows: Business as usual? In *Proc. BPM*, pages 31–47, Ulm, Germany, 2009.
13. Y. Luo et al. Storing and indexing massive RDF datasets. In R. Virgilio et al, editor, *Semantic Search over the Web*, pages 31–60. Springer Berlin, 2012.
14. Management of external data (SQL/MED). ISO/IEC 9075-9:2008.
15. S. Melnik et al. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.
16. J. Roijackers. Bridging SQL and NoSQL. MSc Thesis, Eindhoven University of Technology, 2012.
17. P. J. Sadalage and M. Fowler. *NoSQL distilled: A brief guide to the emerging world of polyglot persistence.* Addison Wesley, 2012.
18. J. Shanmugasundaram et al. A general technique for querying XML documents using a relational database system. *SIGMOD Record*, 30(3):20–26, 2001.
19. UnQL. http://unql.sqlite.org .