Executing SPARQL queries over Mapped Document Stores with SparqlMap-M

Jörg Unbehauen
University of Leipzig
Augustusplatz 11
04109 Leipzig, Germany
unbehauen@informatik.uni-leipzig.de

Michael Martin
University of Leipzig
Augustusplatz 11
04109 Leipzig, Germany
martin@informatik.uni-leipzig.de

ABSTRACT

With the increasing adoption of NoSQL data base systems like MongoDB or CouchDB more and more applications store structured data according to a non-relational, document oriented model. Exposing this structured data as Linked Data is currently inhibited by a lack of standards as well as tools and requires the implementation of custom solutions. While recent efforts aim at expressing transformations of such data models into RDF in a standardized manner, there is a lack of approaches which facilitate SPARQL execution over mapped non-relational data sources. With SparqlMap-M we show how dynamic SPARQL access to non-relational data can be achieved. SparqlMap-M is an extension to our SPARQL-to-SQL rewriter SparqlMap that performs a (partial) transformation of SPARQL queries by using a relational abstraction over a document store. Further, duplicate data in the document store is used to reduce the number of joins and custom optimizations are introduced. Our showcase scenario employs the Berlin SPARQL Benchmark (BSBM) with different adaptions to a document data model. We use this scenario to demonstrate the viability of our approach and compare it to different MongoDB setups and native SQL.

1. INTRODUCTION

Document and NoSQL[1] stores nowadays belong to the standard toolkit when developing web applications. The document data base MongoDB, is ranked as the 4th most popular database by db-engines.com[2] and therefore the most prominent example of a NoSQL database. Consequently, more and more structured data resides in such databases. As of now custom solutions for exposing such data as Linked Data or as SPARQL are required. With SparqlMap-M we demonstrate how we can use existing standards for exposing data. In our previous work [19] and in section 5 we show that SPARQL can be completely translated into SQL. However due to the different data model and limited querying capabilities a direct translation is not feasible for NoSQL Data Base Management Systems (DBMS[3]). The lack of complex query operators like the relational join operator requires different data modeling and retrieval techniques, which pose a challenge to naive mapping and translation approaches. Therefore, in NoSQL DBMS relations are often denormalized and hence data is duplicated. Due to the horizontal scaling capabilities of NoSQL systems, space constraints are loosened and without the need to perform joins NoSQL systems can both achieve higher performance and better scaling characteristics[4]. Mapping such a system into RDF can be achieved by using xR2RML[8], a decalarative mapping language which extends the RDB to RDF Mapping Language (R2RML) [5]. Applied on stores containing duplicate data such a mapping will produce duplicate triples. When materializing, i.e. creating an RDF serialization, of such a mapped graph, this does not pose a problem and will just result in a larger file or require some post processing. Even more simple, duplicated data can be omitted from the mapping. When answering SPARQL queries over a mapped, virtual graph mappings of duplicate data will lead to unnecessary retrieval operations which can seriously degrade performance. With SparqlMap-M we take these duplicates into account and aim at querying only the parts of the data which are relevant for the query.

The rest of the paper is structured as follows. In section 2 we introduce the basic terms, define a running example and propose a relational view on document data bases. This relational view allows us to reuse the same concepts as in relational DBMS. Subsequently, in section 3 we revisit the existing SPARQL translation pipeline and extend it with the required facilities for mapping over a document store. In the following section 4 our approach, SparqlMap-M, is benchmarked and analyzed. Section 5 examines related approaches and similar concepts developed by others. Finally, we draw a conclusion in section 6.

2. NOSQL, RELATIONAL AND DOCUMENT DATA BASE MANAGEMENT SYSTEMS

[1]the term NoSQL is briefly introduced in section 2
[2]http://db-engines.com/en/ranking as of May 2016
[3]We use the terms DBMS, data base and store interchangeably

SEMANTiCS 2016, Leipzig, Germany
© 2016 ACM. ISBN 123-4567-24-567/08/06. . . $15.00
DOI: 10.475/123_4

This chapter briefly introduces the related concepts, previous work and illustrates the problem with a running example.

2.1 Definitions and Scoped Systems

For a clear differentiation of NoSQL, document store and related terms, we follow closely the notions of [4], but focus on the aspects relevant for this paper. In [4], one of the aspects which differentiates NoSQL from relational DBMS is their usage of narrowly scoped retrieval operations and transactions. NoSQL DBMS are further characterized by their horizontal scaling capabilities which are achieved by distributed indexes and storage mechanisms and less rigid schema enforcement. In [4] different data models of NoSQL DBMS are characterized, namely *key-value stores*, *document stores*, *extensible record stores* and *new relational data bases*. For the scope of this paper we focus on document stores, as key-value stores do not allow feature secondary indexes. This lack renders key-value stores unsuitable for executing parts of a SPARQL query as a selective materialization is not possible. As extensible record stores and *new* relational data bases share many characteristics with document stores in terms of data model and data access the concepts and ideas presented in this paper are applicable to them as well. While we do not formally consider them, our implementation is capable of utilizing them as well. Another characteristic of a document store is the lack of a strict schema and data is consequently stored in documents. Documents are key-value maps, where the keys are strings and the values either scalar, lists or documents and are grouped into collections [4].

2.2 Running Example

For a better understanding, we introduce here a running example which we will use throughout the paper. Figure 1 (a) depicts relational tables which describe employees and departments, for example for a small company. Let us assume a user wants to get a list of all employees of the *Research* department. The query in figure 1 (d) executed on a relational DBMS joins the two tables on the foreign key of the person and primary key of the department table and projects all attributes. The where clause enforces the selection of only the employees of the *Research* department.

A direct translation of figure 1 (a) into documents is presented in figure 1 (b). The document store contains two collections, each with documents containing the scalar values derived from the relations. In order to get the list off all employees in the *Research* department, two simple retrieval operations are necessary. Firstly, in figure 1 (e), line 1 the department is retrieved by its name in order to get the *id*. Secondly, with the retrieved *id* in figure 1 (e), line 3 employees are selected. Document stores are optimized for those small scoped operations. However, when queries cannot be expressed in those operators, there are two options. Some document DBMS allow the execution of complex jobs by offering a map/reduce interface[4] or vendor specific facilities such as MongoDBs aggregate pipeline[5].

Algorithm 1 Recursive document to relation translation

```
function asRelation($d$,$prefix$)
    $t \leftarrow \{\}$                                  ▷ Set of Key/Values, a dynamic relat. tuples
    for all $(k, v)$ with $v$ being a scalar values do
        $t \leftarrow t \cup \{(k, v)\}$                  ▷ Adding attributes and values
    end for
    $T \leftarrow \{t\}$
    $T_d \leftarrow \{\}$                                ▷ Contains tuples from nested docs
    for all $(k, v)$ with $v$ being document values do
        $T_d \leftarrow T_d \cup asRelation(v, prefix + k)$
    end for
    $T_d \leftarrow T \times T_d$                        ▷ Join existing tuples with new ones.
    $T_a \leftarrow \{\}$                                ▷ Contains tuples from arrays.
    for all $(k, v)$ with $v$ being unnested array values do
        if $v$ is scalar then                            ▷ As previously with unnesting.
            $T_a \leftarrow T_a \cup \{(k, v)\}$
        else
            $T_a \leftarrow T_a \cup asRelation(v, prefix + k)$
        end if
    end for
    $T_a \leftarrow T \times T_a$
    if $T_d \cup T_a$ is not empty then
        $T \leftarrow T_d \cup T_a$
    end if
    return $T$                                           ▷ All tuples generated from this doc.
end function
```

Running complex queries on distributed stores however can introduce high latency on query evaluation and therefore is used for analytics.

Latency sensitive queries can be efficiently executed on document stores, if the documents are prepared for those queries. In most cases this involves precomputing and aggregating data in the background and denormalizing relations. Figure 1 (c) denormalizes the data from figure 1 (b) by replicating the document instead of only referencing by id. This also simplifies the retrieval operation in figure 1 (f) to a single operation. By increasing data by replication query complexity can be reduced. With their horizontal scaling features, document stores can better deal with the increase of storage volume in traditional relational database systems [4] .

2.3 A relational view on document stores

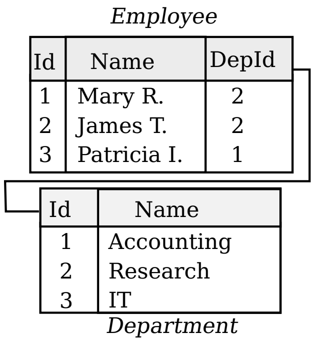
In order to use the same mapping semantics when utilizing a document store we impose a relational view on document stores. Informally, we flatten the tree structure of a document into a table of values, with each column representing a value. The column names are created by concatenating parent keys.

More formally speaking, documents can be mapped into a relation R, with attributes U and a set of tuples T, following the notation introduced in [1]. The conversion is presented in algorithm 1, which defines a recursion starting with a document d of a collection. Due to the schema-free nature of documents in a document store, each document $d_1, d_2, .., d_n$ U_n may be different. We use in algorithm 1 sets of pairs (u, a) to represent relational tuples, U is the union of all keys used during the creation of T.

Exemplary, we calculate the relational view on department `{id:1}` of our running example from figure 1 (c).

```
id |name       |emp.id   |emp.name
---+-----------+---------+----------
2  |Research   |1        |Mary R.
2  |Research   |2        |James T.
```

[4]for example CouchDB http://couchdb.apache.org/
[5]http://mongodb.org



Employee

Id	Name	DepId
1	Mary R.	2
2	James T.	2
3	Patricia I.	1

Id	Name
1	Accounting
2	Research
3	IT

Department

(a) Minimal relational example

```
Employee
{ id: 1,
  name: "Mary R.",
  depid: 2},
{ id: 2,
  name: "James T.",
  depid: 2 },
{ id: 3,
  name: "Patricia I.",
  depid: 1 }
Department
{ id: 1, name: "Accounting" },
{ id: 2, name: "Research" },
{ id: 3, name: "IT"}
```

(b) Naive adaption of the relation model

```
Employee
{ id: 1, name: "Mary R.",
  dep: { id: 2, name: "Research" }},
{ id: 2, name: "James T.",
  dep: { id: 2, name: "Research" } },
{ id: 3, name: "Patricia I.",
  dep: { id: 1, name: "Accounting" } }
Department
{ id: 1, name: "Accounting",
 emp: [{id: 3, name: "Patricia I."}]},
{ id: 2, name: "Research",
 emp: [{id: 1, name: "Mary R."},
     {id: 2, name: "James T."}]},
{ id: 3, name: "IT"}
```

(c) Document with replicated data

```
SELECT *
FROM Employee e
  JOIN Department d
  ON (e.DepId = d.Id)
WHERE d.Name = "Research"
```

(d) SQL query

```
db.department
  .find({name: "Research"})
db.employee
  .find({depid: "2"})
```

(e) Multi-staged document retrieval

```
db.department
  .find({name: "Research"})
```

(f) Single nested document retrieval.

Figure 1: Relational and Document data model representation of the Employee Department use case.

The tuples forming `id` and `name` and their values were determined in algorithm 1 line 2-3 during the first recursion on the root document Department `{id:1}`. Algorithm 1 line 8-10 steps recurses into the nested document, which creates two tuple representing sets, one for each employee. Algorithm 1 line 11 calculates the product of these sets, which are determined in line 21 to be the result.

The resulting relation R does however not fully comply with the relational model, as the domain of the attributes may differ. In R2RML term maps produce for different domains of attributes different literals. For example a mapping of a column of boolean type produces a differently typed literal than an integer typed column. Consequently this functionality cannot be applied on mapped document stores and is therefore omitted.

2.4 Mapping Terminology

In this section we briefly revisit the terminology introduced in [19]. RDF-terminology such as RDF term and SPARQL query operators, such as join, union or triple patterns are used as defined by [11].

Term map: A *term map* is a tuple $tm = (A, ve)$ consisting of a set of relational attributes U from a single relation R and a value expression ve that describes the translation of U into RDF terms (e.g. R2RML templates for generating IRIs). We denote by the range $range(tm)$ the set of all possible RDF terms that can be generated using this term map. Term maps are the base element of a mapping. An example for such a *term map* is in figure 3 (a) line 10 the template `department/{id}`. This term map creates URIs by concatenating the mappings base prefix with "department/" and the content of the attribute "id" from the "department" relation.

Triple map: A triple map *trm* is the triple (tm_S, tm_P, tm_O) of three *term maps* for generating the subject, predicate and object of a triple. All attributes of the three *term maps* must originate from the same relation R. A triple map defines how triples are actually generated from the attributes of a relation (i.e. rows of a table). R2RML allows both SQL tables and views expressed by SQL queries to be used as relation, which is consquently termed *logical table*. Our definition conflicts with the R2RML `TriplesMap` definition, as a triple map always has one subject, one predicate and one object. R2RML on the other hands allows the definition of multiple predicate/object pairs. As this differentiation is only syntactical we use in our examples the R2RML syntax, as it more commonplace. For the rest of this paper we assume an implicit transformation into single predicate/object map `TriplesMaps` and use triple map for definition purposes. The example figure 3 indicates triple maps in a `TriplesMaps`.

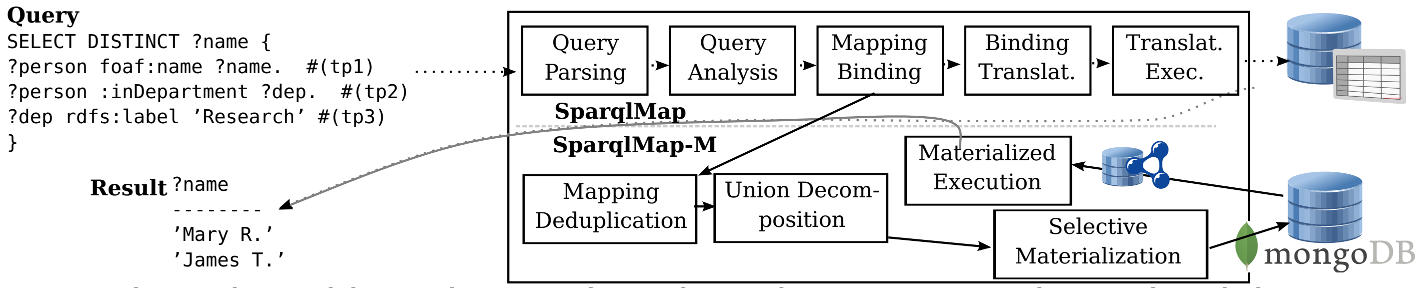
3. SPARQLMAP-M: SPARQL OVER DOCUMENT STORES

In the following section we briefly review the existing SPARQL to SQL translation pipeline of SparqlMap [20] and show the required extensions for dealing with document stores. An overview over the existing pipeline and the additional steps are is given in figure 2. To illustrate the mapping process, we use the SPARQL translation of our initial SQL query of figure 1.

Query Normalization The query is transformed into an algebraic query tree and normalized, such that uniform access for all subsequent optimizations and transformations is provided.

Query Analysis The scope of the filters and variables is determined and query operators are summarized.

Mapping Binding In a recursive pre-evaluation of joins and filters, the number of potential triple maps is reduced. The result of this step is a query binding, a set of tuples, which associates each triple pattern of a query with the set of candidate triples maps, denoted $(tp_1 : tm_1, ..., tm_n)$. For each each triple pattern in the query, in the beginning



Query

```
SELECT DISTINCT ?name {
?person foaf:name ?name.  #(tp1)
?person :inDepartment ?dep.  #(tp2)
?dep rdfs:label 'Research' #(tp3)
}
```

Result

```
?name
--------
'Mary R.'
'James T.'
```

Figure 2: The SparqlMap and the SparqlMap-M pipelines with a sample SPARQL query over the mapped sample data set.

of the mapping binding step all triple maps are considered candidates. By iteratively applying filters and joins on the bindings, the number of candidate triple maps is reduced. In the end ideally only the triple maps, which contribute to the result set of the query are left in the binding.

Binding Translation In another bottom up recursion over the query, the SPARQL query operators are translated into equivalent SQL operators. The relations and attributes, on which the SQL operators are performed upon are determined by the bindings. Further, during this step optimizations for efficient filter construction, join and left join handling and union prevention take place, as we describe in [20].

Query Execution Finally the translated SQL query is executed over the mapped document store. The SQL result set is syntactically transformed into a SPARQL result set and presented to the client.

As we depict in figure 2 SparqlMap-M utilizes the mapping binding to perform additional steps required to execute SPARQL over document stores. The main differences of document stores compared to relational DBMS are (a) the lack of both union and JOIN operators and (b) the presence of duplicate data.

In the following we present our solutions of these issues. Regarding (b) we extend R2RML mappings with declarative means for indicating denormalized data and propose an algorithm for removing them. For dealing with (a), we perform a multi-level query execution, materializing selected parts of the mapped data in an internal store and subsequent execution the remaining query over it.

3.1 Mapping Deduplication

As we already introduced in section 2, document stores replicate data to achieve high performance instead of retrieving data from multiple collections. While this helps to create better queries for the document DBMS user, this poses a challenge for a SPARQL execution over document stores. To illustrate this problem, we perform a step by step translation using the SPARQL representation of the

$$\text{(a) } \{(tp_1, \{tm_1\}), (tp_2, \{trm_2\}), (tp_3, \{trm_3\})\}$$

$$\text{(b) } \{(tp_1, \{trm_1, trm_4\}), (tp_2, \{trm_2, trm_5\}), (tp_3, \{trm_3\})\}$$

$$\text{(c) } \{(tp_1, \{trm_1\}), (tp_2, \{trm_4\}), (tp_3, \{trm_5\})\}$$

Figure 4: Binding for (a) naive mapping (b) mapping on duplicate data and (c) deduplicated mapping

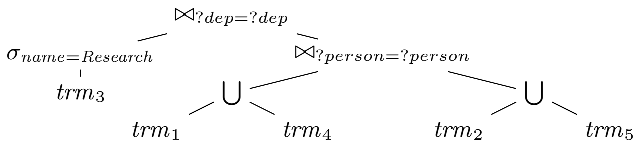


Figure 5: Query plan for nested unions

running example query figure 1. This SPARQL query is depicted in figure 2 and consists of three triples patterns. In the following we will showcase the mapping binding and translation in three cases. First, a naive document store with a mapping. Second, a document store with duplicates and a non-duplicate aware mapping and third, a document store with duplicates and a duplicate aware mapping.

Naive document store with mapping.

In this scenario we utilize the mapping in figure 3 (b) for translating the naive document store of 1 (b). An execution of the mapping binding step of the SparqlMap pipeline creates the mapping bindings in figure 4 (a). This mapping binding associates each triple pattern with a single term map. The query and binding executed over a relational database, creates a SQL translation that features a single, efficient query, with a SQL-join between the logical tables of trm_3 and tm_1. tm_1 creates a self join, that can be optimized, as we illustrate in [20]. As document stores cannot execute joins, the same query and binding on a document store would require a complete materialization of trm_1 and trm_2. As trm_1 and trm_2 share the same logical table, so only one materialization has to be performed. Still, the resulting performance would be subpar and this behavior can be observed in the evaluation for the SparqlMap-M-Naive scenario, section 4.3.

Document store with duplicates and a non-duplicate aware mapping.

When using replication in document DBMS to increase performance, these duplicates have to be mapped. This mapping is expressed by figure 3 (a) and (b) and is applied on the document store with data replication of 1 (c). Note, that in figure 3 (a) line 7 and 8 are not interpreted, as this is the replication declaration we showcase in the third scenario. Executing the query of figure 2 of such a mapped document store yields the mapping binding presented in figure 4. The query translation process defined in [19] would consequently yield the query plan in figure 5.

As document stores cannot execute unions and joins ef-

```
<TriplesMapDepartmentEmployee>
  sm:replicateOf <TriplesMapEmployee>;
  sm:replicateIn <TriplesMapDepartment>;
  rr:logicalTable [rr:tableName "Department"];
  rr:subjectMap [rr:template "employee/{emp.id}"];
  rr:predicateObjectMap [ # triple map trm4
    rr:predicate foaf:name;
    rr:objectMap [rr:column "emp.name"]];
  rr:predicateObjectMap [ # triple map trm5
    rr:predicate <inDepartment>;
    rr:objectMap [rr:column "id"]].
```

(a) Additional duplicate mapping.

```
<TriplesMapEmployee>
  rr:logicalTable [rr:tableName "Employee"];
  rr:subjectMap [rr:template "employee/{id}"];
  rr:predicateObjectMap [ # triple map trm1
    rr:predicate foaf:name;
    rr:objectMap [rr:column "name"]];
  rr:predicateObjectMap [ # triple map trm2
    rr:predicate <inDepartment> ;
    rr:objectMap [
      rr:template "department/{depid}"]].
<TriplesMapDepartment>
  rr:logicalTable [rr:tableName "Department"];
  rr:subjectMap [rr:template "department/{id}"];
  rr:predicateObjectMap [ # triple map trm3
    rr:predicate rdfs:label;
    rr:objectMap [rr:column "name"]].
```

(b) R2RML mapping of figure 1 (c)

Figure 3: R2RML mapping for figure 1.

fectively, a direct execution leads to low performance and requires extensive materialization of the collections. To make matters worse, the documents retrieved from the DBMS contain both the replicated as well as the original data. Just by examining the execution plan, it is clear that even though the document store contains replicated data to improve performance, a naive query translation over this data store would be slower than without the replication. This scenario is labeled SparqlMap-M-Dup in the evaluation.

Document store with duplicates and duplicate aware mapping.

For efficent utilization of duplicates in a SPARQL over document DBMS scenario, the query processor needs to be aware of duplicates. With mapping deduplication we remove triple maps, that would provide in the context of a query duplicate results. In SparqlMap-M this is achieved by annotating R2RML mappings, as shown in figure 3. We utilize here the `sm:replicateOf` and `sm:replicateIn` to declare, that a triple map is used to translate replicated data into RDF. `sm:replicateOf` references here term map, which holds original data and `sm:replicateIn` declares into which collection the term map was replicated.

Mapping deduplication is therefore a recursive query preevaluation, in which triple maps are removed from mapping bindings. Assume a given mapping binding for two triple patterns tp_1 and tp_2, each with a separate set of triple maps TM_1 and TM_2. For each join operator with a single shared variable v between the triple patterns, we can remove a triple map $trm \in TRM_2$, if there exists a $trm_r \in TRM_2$ for which holds:

- v is in the subject position of tp_1 and in the object position tp_2 and
- There exists a term map $trm_{rRep} \in TM_1$ that has the same logical table as trm_r.
- trm is the original term map of trm_r

In a second step, we remove all replicas from all bindings, in which also the original is present. Consequently, after applying mapping deduplication, the mapping binding of figure 2 is the binding presented in figure 4 (c). As all triple maps originate from the same collection, this query can be expressed in an efficient simple lookup. In

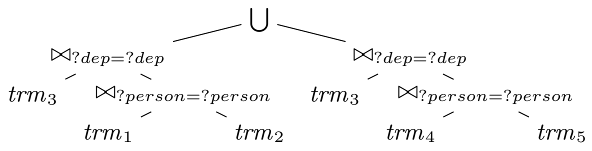


Figure 6: Query plan with pushed union

the evaluation this scenario corresponds to the SparqlMap-M-DupAware benchmark run.

Duplicate removal is based on triple maps, but in the sample in 3 we use `sm:replicateOf` on an R2RML triples maps. Based on the definition above, the properties need to target individual triple maps. We assume therefore the implicit translation that based on matching predicates this annotation is conveyed to the individual triple maps. This restricts the usage of duplicate indicating to `TripleMaps` with locally unique properties. We therefore consider the mapping extension preliminary and are still exploring how we can indicate duplicates in a user friendly way.

3.2 Union Decomposition

Besides unions occurring in SPARQL queries, unions are used in SparqlMap to translate mapping bindings with more than one triple map per triple pattern. In the SQL translations, the unions are used in conjunction with joins, the union operator cannot be performed in the underlying database. Therefore a SPARQL query is translated into union free sub-queries. In a first translation step, all union operators of a query are pushed to the top of the query using the transformations defined in [15]. In a second step, we transform the current form of a mapping binding. As described in [19] we translate mapping bindings with more than one triple map into a union. In order to produce as large as possible union free subqueries, we compute a nested query plan using the previously computed bindings figure 5. This nested query plan can be transformed again by pushing the unions to the top of the algebraic query tree as depicted in figure 6.

3.3 Selective Materialization and Query Execution

Document stores do not feature union and joins operators[4]. For facilitating SPARQL query execution these

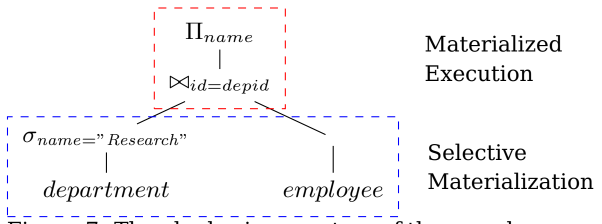


Figure 7: The algebraic operators of the sample query (figure 2 aligned to the SparqlMap-M pipeline.

operators have to be executed outside of the DBMS. Using the transformed query of the union decomposition step, we construct union and joins free subqueries. A simplified version of the sample query of figure 2 is presented in figure 7. The selections in this sample can be pushed into the underlying database, which corresponds to figure 2 Selective Materialization. The results of those queries are materialized into an internal RDF store. In this materialization step we prevent duplicates, by a simple check. This check examines if a collection has already been fully materialized and prevents the repeated materialization of this collection or any subsets of that collection. The previously unprocessed query operations are than executed on this internal RDF store.

4. EVALUATION

For our evaluation we briefly introduce our implementation and define the benchmark setup. The results are subsequently discussed.

4.1 Implementation

Our prototypical implementation of SparqlMap-M is available on github[6]. SparqlMap-M can be used acting as both command line tool or SPARQL endpoint. As SparqlMap-M builds on Apache Jena[7][7] for SPARQL analysis and parsing, we utilize Jena in-memory models for the selective materializing step. Further, the relational abstraction over the document store is provided by Apache MetaModel[8], which provides a pseudo-relational view on both relational and NoSQL DBMS. Apache Metamodel performs the joins operations, the underlying DBMS is unable to perform. However, as of now, MetaModel does not feature advanced remote query execution facilities as the focus is on schema discovery and uniform data access. This contrast MetaModel to federated SPARQL query execution engine like ANAPSID [2], which feature sophisticated query planners.

4.2 Benchmark Setup

We evaluated our approach using a customized version[9] of the Berlin SPARQL Benchmark (BSBM). The data set of BSBM is synthetic and features three different use cases, all of which represent interactions with a product database. The entites are modeled after consumer $products$, which have certain $productFeatures$ and other properties and relations. A thorough description of the benchmark can be found in [3]. The BSBM as a SPARQL Benchmark is used for both evaluating RDF as well as relational database management system as it allows the creating of data sets in either SQL or RDF serialisations and has SQL translations of its SPARQL queries. In our effort to evaluate SparqlMap-M, we designed two translations of the SQL output into MongoDB databases: MongoDB-Naive and MongoDB-Dup. We also created R2RML mappings of the databases for three mappings cases: SparqlMapl-M-Naive, mapping the MongoDB-Naive database. SparqlMap-M-Dup, mapping MongoDB-Dup, but without indicating duplicates and SparqlMap-M-DupAware, which extends SparqlMap-M-Dup with duplication indicators. MongoDB-Naive is a naive translation from the relational model similiar to the translation from figure 1 (a) into figure 1 (b) . For each of the ten tables in the SQL output of BSBM, a collection in a MongoDB instance is created and populated with the relational entities as documents without any nested documents. Consequently, the n:m relation between a $product$ and $productFeatures$ is represented in a separate collection. As the queries consequently have to collect data from multiple collections, this schema does not represent how a real world document store is utilized. It however allows us to test the performance translated queries, when the schema and queries are not well aligned.

For MongoDB-Dup data set we denormalize parts of the relational data into documents in order to reduce the number of queries for satisfying the information needs of the original BSBM explore queries. A relation denormalized here is for example (i)$product \Leftarrow productFeature$ or (ii)$offer \Leftarrow product$. The product collection size grew in case of (i) from 43.4 MBs to 156.90 MBs and in duplicating (ii) $product$ into $offer$, the collection size increased from 148.42 MBs to 1641.06 MBs, as reported by MongoDB. Like previously performed on the relational data set [20], we added indexes on all attributes used as selectors in queries. Further, for each of MongoDB-Naive and MongoDB-Dup we translated the BSBM queries into either find operations or aggregate pipelines, such that the output contains the same data[10]. All tests were performed on an Intel Xeon E3-1220 server with 8 GB of RAM with all indexes and data is residing on an 128 GB SSD running an Unbuntu Server 15.10.

Each of the database system, i.e. MongoDB 3.2 and PostgreSQL 9.0 was assigned 2 GB RAM and each system was restricted to only use a single core, further we used no clustering or similar features. We virtualized all data stores using Docker[11]. To ensure complete saturation of the system under test we used 4 client threads in parallel and otherwise resorted to default values. The scale factor for BSBM was 27850, which yields about 10 million triples.

4.3 Benchmark Analysis

The results from our benchmark runs are displayed in figure 8. We will first discuss the results of the native data stores and then relate them to the SparqlMap-M results.

[6]http://github.com/tomatophantastico/sparqlmap
[7]http://jena.apache.org
[8]http://metamodel.apache.org
[9]https://github.com/tomatophantastico/bsbm
[10]queries and benchmark are available on http://github.com/tomatophantastico/bsbm
[11]https://github.com/Iranox/bsbmloader-docker

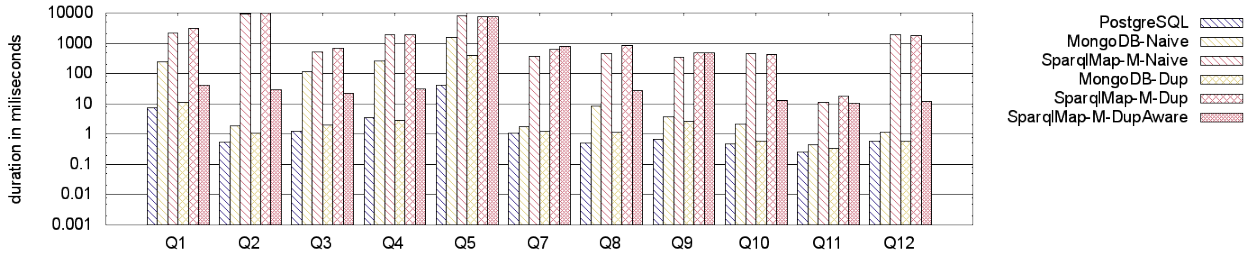


Figure 8: Query execution time of BSBM queries on a 10 million triples data set.

Overall, PostgreSQL is fastest, followed by the MongoDB-Dup scenario. The MongoDB-Naive scenario was in most cases not able to compete. MongoDB-Dup utilizes in all but Query (Q) 5[12], Q7 and Q9 simple document retrieval operations and can therefore in some cases surpass PostgreSQL. Q4 is one of the cases where PostgreSQL has to perform several joins, which is more time consuming than the simple retrieval in MongoDB-Dup. The advantage of PostgreSQL comes apparent in Q5, which is generally the most expensive query to execute as it features a large self join. In the MongoDB cases Q5 was executed as `lookup` in the aggregate pipeline. While MongoDB-Dup can reduce the number of collections used and is about 4 times faster than MongoDB-Naive it remains about 10 times slower than native PostgreSQL. Q7 is an example for effective usage of simple document retrieval operations. This query can be expressed as series of look up in MongoDB-Naive and therefore the performance is very close to both PostgreSQL and MongoDB-Dup. Similar simple find operations also define Q2 and Q9-Q12, consequently their performance is close to their SQL equivalents. However, as a lot of queries are of a discovery type like Q1, which queries for products with certain features, performance degrades as data needs to be aggregated in a query pipeline first or be evaluated by the client in memory. We tested both client-side evaluation, which means that partial query results are simply looped and additional resources are retrieved and expressing queries as `aggregate` pipelines, with similar results. In all cases they perform slower than their MongoDB-Dup or PostgreSQL counterparts.

As a general note, SparqlMap-M performs intensive query processing and evaluation. Therefore we can observer a constant overhead of at least 10ms per query. Q11 is the query, where in all cases the same amount of data is fetched from DBMS. But due to overhead and the extremely fast query processing on the native stores, this constant overhead results in a 20 fold query execution time increase.

The SparqlMap-M-Naive scenario is best compared with the MongoDB-Naive scenario. Queries that are highly selective on data from multiple collections can very well by implemented by multiple client fetch. As SparqlMap-M cannot perform these multi-step fetch, Q2 requires a full materialization of related collections, resulting in low performance. This also applies to Q7, Q9 and Q10 and Q12. For less selective queries, for example Q1, Q3, Q4 and Q8, the gap closes, however document retrieval is still less efficient and expensive. Q5 requires in all cases intensive query processing is the most demanding query, requiring multiple full materialization of collections.

SparqlMap-M-Dup in shows a similar performance as SparqlMap-Naive, as the query processor cannot remove duplicates. In a comparison with MongoDB-Dup, the differences in query performance become even more apparent. Whereas MongoDB-Dup increases performance, in most cases SparqlMap-M-Dup is even slower, due to the higher amounts of data retrieved from the document store.

SparqlMap-M-DupAware shows how Mapping Deduplication can increase query performance. Queries that previously required full materializations of collections can now be answered using a single find (Q1-Q4, Q8,Q9-Q12). We can observe several orders of magnitude in query performance here. For virtually all other queries, no improvements in query speed can be observed, as these still require materialization of not replicated collections.

5. RELATED WORK

Translating the SPARQL to SQL is a well studied technique for accessing relational databases. Examples of those rewriters are ontop [13], morph [12], sparqlify[17] and ultrawrap[16], however none of those rewriters support document DMBS. R2RML [5] is a W3C standard for relational to RDF mappings and is supported by most of these rewriters. Our work is the continuation of such an SPARQL to SQL rewriter. Conceptually, SparqlMap-M shares with D2R the idea of not pushing all operations into the underlying DBMS, but to perform them in the rewrite engine. This contrast with most other approaches, which aim at executing the whole query in the underlying database.

Mapping other structured sources like JSON or XML into RDF has a long track record, which is for example surveyed in [18]. A fresh take in translating data in document data stores is xR2RML[8], an extension of the R2RML mapping language. These approaches extends the R2RML with concepts for accessing and matching parts of documents in non-relational datastores or non-relational data, like JSON contained in relational databases. This contrasts with SparqlMap-M which only minimally extends R2RML by denoting duplicate *rr:PredicateObjectMap*s and only provides simple matching capabilities using a pseudo-relational view on document data.

The problem of duplicate data is also relevant for SPARQL federation engines. A similar problem arises here: Without efficient remote join capabilities, potentially huge result sets have to be transferred to perform the join locally.

[12] BSBM queries are described here: http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark

Approaches like DAW [14] and FEDRA [9] aim at reducing the amount of data to be transferred by examining the data (DAW) or by declaring replication sets (FEDRA). The idea of replicated fragments of FEDRA is very closely related the replications encountered in document stores as both are declaratively provided and both are used to reduce the amount of selected sources or respectively *predicateObjectMap*s.

Another interesting work is D-SPARQ [10], a distributed RDF store, which uses the MongoDB as storage solution. The authors perform a different approach here, as they update the data model to the expected input queries. With SparqlMap-M however the data storage is considered user provided and an adoption might interfere with the main application the document store is serving.

An overview and categorization of NoSQL and their (potential) usage in Linked Data applications is given in [6]. The approaches and technologies surveyed here focus like D-SPARQ on using them as building blocks for Linked Data applications and not on how to integrate systems using these cloud based technologies into a Linked Data environment.

6. CONCLUSIONS AND FUTURE WORK

With SparqlMap-M we showcase that we can execute SPARQL queries on remote document stores without creating an RDF dump first. We demonstrated that using a simple and straight forward R2RML extension we can mitigate the query rewriting issues encountered with duplicated data in document DBMS. Generally speaking, the shape of the data and the shape of the queries ultimately determine query execution time, especially in the case of document stores. We do not think that execution of arbitrary SPARQL queries over a mapped document can achieve levels of performance close to a dedicated triple store. The join operator is a core feature of SPARQL and document stores have limited support for executing a join. However, we think the achieved performance is sufficient for most scenarios, safe for scenarios with high requirements on load tolerances and low latency. Therefore exists no need for creating synchronizations and no additional backend server is required. SparqlMap-M can act just as an additional web front end on a document database.

Our future work lies in further reducing the cost of executing SPARQL queries over mapped document stores. An effort to reduce the overhead of materializing whole collections is the implementation of advanced joins and a deep integration into the Apache Jena query processing framework. In future work an adaption of the agjoin described in [2] offers the potential to increase performance and decrease the memory footprint. Additionally, we will look into new query languages for NoSQL stores like Couchbase N1QL that allow joining over multiple collections and further examine next-gen multi-model databases like ArangoDB.

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[2] M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. Anapsid: An adaptive query processing engine for sparql endpoints. In *ISWC*. 2011.

[3] C. Bizer and A. Schultz. The berlin sparql benchmark. *IJSWIS*, 2009.

[4] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[5] S. Das, S. Sundara, and R. Cyganiak. R2rml: Rdb to rdf mapping language. Technical report, 2012.

[6] M. Hausenblas, R. Grossman, A. Harth, and P. Cudré-Mauroux. Large-scale linked data processing-cloud computing to the rescue?. *CLOSER*, 2012.

[7] B. McBride. Jena: A semantic web toolkit. *IEEE Internet computing*, 2002.

[8] F. Michel, L. Djimenou, C. Faron-Zucker, and J. Montagnat. xr2rml: Non-relational databases to rdf mapping. Technical report, 2015.

[9] G. Montoya, H. Skaf-Molli, P. Molli, and M.-E. Vidal. Federated sparql queries processing with replicated fragments. In *ISWC*. 2015.

[10] R. Mutharaju, S. Sakr, A. Sala, and P. Hitzler. D-sparq: distributed, scalable and efficient rdf query engine. In *ISWC Posters & Demos*, 2013.

[11] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *ISWC*, 2006.

[12] F. Priyatna, O. Corcho, and J. Sequeda. Formalisation and experiences of r2rml-based sparql to sql query translation using morph. In *WWW*, 2014.

[13] M. Rodrıguez-Muro, J. Hardi, and D. Calvanese. Quest: efficient sparql-to-sql for rdf and owl. In *ISWC*, 2012.

[14] M. Saleem, A.-C. N. Ngomo, J. X. Parreira, H. F. Deus, and M. Hauswirth. Daw: Duplicate-aware federated query processing over the web of data. In *ISWC*. 2013.

[15] M. Schmidt, M. Meier, and G. Lausen. Foundations of sparql query optimization. In *ICDT*, 2010.

[16] J. F. Sequeda and D. P. Miranker. Ultrawrap: Sparql execution on relational data. *Journal of Web Semantics*, 2013.

[17] C. Stadler, J. Unbehauen, P. Westphal, M. A. Sherif, and J. Lehmann. Simplified rdb2rdf mapping. In *LDOW*, 2015.

[18] J. Unbehauen, S. Hellmann, S. Auer, and C. Stadler. Knowledge extraction from structured sources. In *Search Computing*. Springer, 2012.

[19] J. Unbehauen, C. Stadler, and S. Auer. Accessing relational data on the web with sparqlmap. In *JIST*. 2012.

[20] J. Unbehauen, C. Stadler, and S. Auer. Optimizing sparql-to-sql rewriting. In *IIWAS*, 2013.

8. ACKNOWLEDGMENTS

This work was partly supported by the following grants from the German Federal Ministry of Education and Research (BMBF) for the LEDS Project under grant agreement No 03WKCG11C and the European Union's Horizon 2020 research and innovation programme for the SlideWiki Project under grant agreement No 688095.