Querying Heterogeneous Document Stores

Hamdi Ben Hamadou[1], Faiza Ghozzi[2], André Péninou[3] and Olivier Teste[3]
[1]*Université de Toulouse, UT3, IRIT (CNRS/UMR 5505), Toulouse, France*
[2]*Université de Sfax, ISIMS -MIRACL, Sfax, Tunisie*
[3]*Université de Toulouse, UT2J, IRIT (CNRS/UMR 5505), Toulouse, France*

Keywords: Information Systems, Document Stores, Structural Heterogeneity, Schema-independent Querying.

Abstract: NoSQL document stores offer support to store documents described using various structures. Hence, the user has to formulate queries using the possible representations of the desired information from different schemas. In this paper, we propose a novel approach that enables querying operators over a collection of documents with structural heterogeneity. Our work introduces an automatic query rewriting mechanism based on combinations of elementary operators: project, restrict and aggregate. We generate a custom dictionary that tracks all representations for attributes used in the documents. Finally, we discuss the results of our approach with a series of experiments.

1 INTRODUCTION

Document-oriented stores are becoming very popular because of their simple and efficient ways to manage large semi-structured data sets. Each record, usually formatted in JSON, is stored inside a document in schema-less fashion. So, a collection groups a heterogeneous set of documents for which no common schema is required. Although this flexibility is very power-full at loading time, the resulting heterogeneity presents a serious issue during querying phase. Indeed, in order to obtain relevant results, users have to be aware of all existing schemas while formulating their queries and have to combines all the schemas in complex queries. Three classes of heterogeneity can be considered in the context of document stores. (Shvaiko and Euzenat, 2005)

- *Structural heterogeneity* points to the different structures that exist in documents. The main issue is the existence of several paths to access the same attribute; e.g., the position of an attribute denoted *"name"* may not the same in two documents (nested, flat).
- *Syntactic heterogeneity* exists when different attributes refer to the same concept; e.g., the attribute *"name"* may be denoted *"name," "names"* or *"first_name"* in different documents.
- *Semantic heterogeneity* exists when the same attribute refers to different concepts; e.g., the attribute *"name"* may designate a *"person name"*, an *"animal name"* or a *"disease name"* depending on documents.

In this paper, we focus on the structural heterogeneity issue in document stores.

Example. We use the example collection of figure 1 composed of five documents describing authors and some of their publications. Documents are described using JavaScript Object Notation (Bourhis et al., 2017).

Let us suppose we are interested in collecting information related to *"name of authors"* and their publications. The query will be formulated over the attributes *"name"* and *"title"*. Any user may expect results for the five authors of the example (except perhaps for *"paul verlaine"*) and possibly five titles. If we look at figure 1, the attribute *"name"* does not present any problem since it is always in the same position in the five documents. However, the attribute *"title"* may cause some issues because of its various structural positions within the documents. To reach the attribute *"title"* various paths exist in the different document schemas: *"title," "book.title," "artwork.1.title"* and *"artwork.2.title"* (here *".1."* and *".2."* stand for the indexes in the array *"artwork"*).

When using MongoDB data store system, we can formulate the query *db.C.find({}, {"name" : 1,*

Ben Hamadou, H., Ghozzi, F., Péninou, A. and Teste, O.
Querying Heterogeneous Document Stores.
DOI: 10.5220/0006777800580068
In *Proceedings of the 20th International Conference on Enterprise Information Systems (ICEIS 2018)*, pages 58-68
ISBN: 978-989-758-298-1

```
[ { name:"victor hugo",
    title:"les miserables",
    year:1862
  },
  { name:"honore de balzac",
    book:{title:"le pere Goriot",
          year:1835
         }
  },
  { name:"paul verlaine",
    birthyear:1844
  },
  { name:"charles baudelaire",
    artwork:[
      {title:"les fleurs du mal",
       year:1857
      },
      {title:"le spleen de Paris",
       year:1855
      }
    ]
  },
  { name:"pierre de ronsard",
    title:"les amours",
    year:1557
  }
]
```

Figure 1: Collection *"C"* of five example documents.

"title" : 1}*)*. Executing such query will return the following incomplete set of documents because of the structural heterogeneity of the attribute *"title"*:

```
[ { name:"victor hugo",
    title:"les miserables"     },
  { name:"honore de balzac"    },
  { name:"paul verlaine"       },
  { name:"charles baudelaire"  },
  { name:"pierre de ronsard",
    title:"les amours"         } ]
```

If we formulate an alternative query that matches with another path of *"title"*, *db.C.find(* {}, {*"name" : 1, "book.title" : 1*}*)*, the following incomplete set of documents is returned:

```
[ { name:"victor hugo"            },
  { name:"honore de balzac",
    book:{title:"le pere Goriot"} }
  { name:"paul verlaine"          },
  { name:"charles baudelaire"     },
  { name:"pierre de ronsard"      } ]
```

We can notice that each query returns a part of the expected result (author/title pairs) meanwhile returning redundant incomplete results. Moreover, without any other query, these incomplete two queries results can lead the user to interpret that *"charles baudelaire"* has no publication in the collection; and that is not true.

In the literature, two approaches are developed to deal with structural heterogeneity. The data integration approach consists in transforming data according to a unified schema to form a homogeneous collection (Tahara et al., 2014). The automated schema discovery approach provides the various schemas at users (Wang et al., 2015). The data integration may be a time-consuming task because it implies to define the mapping for every variation of schemas, while the automated schema discovery requires that users handle many structures and manage heterogeneity by themselves.

Our approach is designed to resolve these issues. It lets the user query a collection using one schema of some documents, and our system transparently rewrites the user query to take into account all existing schemas. We develop a system that we call *EasyQ* (Easy Query for NoSQL databases), which consists of a schema-independent querying on heterogeneous documents describing a given entity in document-oriented stores. We opt for a solution based on virtual data integration in which we introduce a data dictionary that runs in transparent way and hides the complexity of building the expected queries (Yang et al., 2015).

This paper is organized as follows. The second section reviews the most relevant works that deal with querying heterogeneous documents. Section 3 explains the proposed approach and proposes the formalization of the approach. Section 4 presents our first experiments and the time/size cost of our approach regarding the size of collections and the variety of schemas.

2 RELATED WORK

The problem of querying heterogeneous data is an active research domain studied in several contexts such as data-lake (Hai et al., 2016), federated database (Sheth and Larson, 1990), data integration, schema matching (Rahm and Bernstein, 2001). We classify the state-of-the-art works into four main categories regarding the solution given to handle the heterogeneity problems.

Schema Integration. The schema integration process is performed as an intermediary step to facilitate the query execution. In their survey paper, (Rahm and Bernstein, 2001) presented the state-of-the-art techniques used to automate the schema integration process. Matching techniques can cover schemas or even instances. Traditionally, lexical matches are used to handle the syntactic heterogeneity. Furthermore, thesaurus and dictionary are used to perform semantic matching. The schema integration techniques may lead to data duplication and possible initial underlying data structure loss, which may be impossible or unacceptable to support legacy applications. Let us notice that we built our schema-independent querying upon the ideas developed in schema level matching techniques.

Physical Re-factorization. Several works have been conducted to enable querying over semi-structured data without any prior schema validation or restriction. Generally, they propose to flatten XML or JSON data into a relational form (Chasseur et al., 2013) (Tahara et al., 2014), (DiScala and Abadi, 2016). SQL queries are formulated based on relational views built on top of the inferred data structures. This strategy suggests performing heavy physical re-factorization. Hence, this process requires additional resources such as the need for external relational database and extra efforts to learn the unified inferred relational schema. Users dealing with those systems have to learn new schemas every time they change the workload, or new data are inserted (or updated) in the collection because it is required to re-generate the relational views and the stored columns after every change.

Schema Discovery. Other works propose to infer implicit schemas from semi-structured documents. The idea is to give an overview of the different elements present in the integrated data (Baazizi et al., 2017) (Ruiz et al., 2015). In (Wang et al., 2015) the authors propose summarizing all documents schema under a skeleton to discover the existence of fields or sub-schema inside the collection. In (Herrero et al., 2016) the authors suggest extracting collection structures to help developers while designing their applications. The heterogeneity problem here is detected when the same attribute is differently represented (different type, different position inside documents). Schema inferring methods are useful for the user to have an overview of the data and to take the necessary measures and decisions during application design phase. The limitation with such logical view is the need to manual process while building the desired queries by including the desired attributes and their possible navigational paths. In that case, the user is aware of data structures but is required to manage heterogeneity.

Querying Techniques. Others works suggest resolving the heterogeneity problem by working on the query side. Query rewriting (Papakonstantinou and Vassalos, 1999) is a strategy to rewrite an input query into several derivations to overcome the heterogeneity. The majority of works are designed in the context of the relational database where heterogeneity is usually restricted to the lexical level only. Regarding the hierarchical nature of semi-structured data (XML, JSON documents), the problem of identifying similar attributes is insufficient to resolve the problem of querying documents with structural heterogeneity. To this end, the keyword querying has been adopted in the context of XML (Lin et al., 2017). The process of answering a keyword query on XML data starts by identifying the existence of the keywords within the documents without the need to know the underlying schemas. The problem is that the results do not consider the heterogeneity in term of attributes but assume that if the keyword is found so document is adequate and has to be returned to the user. Other alternatives to find different navigational paths leading to the same attribute is supported by (Clark et al., 1999), (Boag et al., 2002). Only the structural heterogeneity is partially addressed. There is always a need to know the underlying document structures and to learn a complex query language. In addition, these solutions are not built to run over large-scale data. In addition, we notice the same limitations considerations with JSONiq (Florescu and Fourny, 2013) the extension to XQuery designed to deal with large-scale semi-structured data.

This paper takes these ideas one step further by introducing a schema-independent querying approach that is built over the native operators supported by document stores. We believe that, in collections of heterogeneous documents describing a given entity, we are able to handle the documents heterogeneities via the use of query rewriting mechanisms introduced in this paper. Our approach is performed in a transparent way over the initial document structures. There is no need to perform heavy transformation nor to use further auxiliary systems.

3 EASY QUERY FOR NoSQL DOCUMENT STORES

EasyQ is a tool that facilitates to the user the exploratory querying of a document store without having to know the entire data structures of documents.

The figure 2 gives a high-level viewpoint of our engine, divided into two parts: a dictionary builder and a query rewriting engine. To ensure an efficient query enrichment, we introduce *EasyQ* in early stages during data loading phase in order to generate and materialize a dictionary containing all different navigational paths for all attributes. From a general point of view, the dictionary is updated each time a document is updated, removed or inserted in the collection. At the querying stage, *EasyQ* takes as input the user query, called Q, formulated over fields and/or sub-paths, and the desired collection. The *EasyQ* rewriting engine reads from the dictionary and produces an enriched query supported by the underlying document store, called *Qext*. Finally, the document store returns the results to the user.

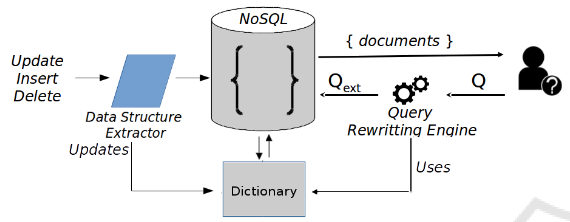

Figure 2: EasyQ architecture: data structure extractor and query rewriting engine.

In the remainder of this section, we describe the formal data model and the extended query process.

3.1 Formal Data Model

Usually, a document-oriented store is modelled as a collection of JSON documents.

Definition 1 *(Collection).* A collection C is defined as a set of documents:

$$C = \{d_1,\ \dots,\ d_c\}$$

Definition 2 *(Document).* A document d_i, $\forall i \in [1,c]$, is defined as a *(key,value)* pair:

$$d_i = (k_i, v_i)$$

- k_i is a key that identifies the document;
- $v_i = \{a_{i,1} : v_{i,1}, \dots, a_{i,n_i} : v_{i,n_i}\}$ is the document value. The document value v_i is defined as an *object* composed by a set of *($a_{i,j}$, $v_{i,j}$)* pairs, where each $a_{i,j}$, is a *string* called *attribute* and each $v_{i,j}$, is the *value* that can be *atomic* (numeric, string, boolean, null) or *complex* (object, array).

An atomic value is defined as follows:

- $v_{i,j} = n$ if $n \in \mathbb{N}^*$, the set of numeric values;
- $v_{i,j} =$ "s" if "s" is a string formatted in Unicode characters of Σ^*;
- $v_{i,j} = b$ if $b \in \mathbb{B}$ the set of boolean values $\mathbb{B} = \{true, false\}$;
- $v_{i,j} = \perp$ is a null value.

A complex value is defined as follows:

- $v_{i,j} = \{a_{i,j,1} : v_{i,j,1},\ \dots,\ a_{i,j,n_{i,j}} : v_{i,j,n_{i,j}}\}$ is an object value where $v_{i,j,k}$ are strings formatted in Unicode characters of Σ^* called *attributes* and $v_{i,j,k}$ are values; This is a recursive definition identical to document value;
- $v_{i,j} = [v_{i,j,1},\ \dots,\ v_{i,j,n_{i,j}}]$ is an array of values.

In case of having document values $v_{i,j}$ as an object or array, their inner values $v_{i,j,k}$ can be complex values allowing to have different nesting levels. To cope with nested documents and navigate through schemas, we adopt classical navigational path notation (Bourhis et al., 2017).

Definition 3 *(Schema).* A schema, denoted s_i, inferred from the document value $\{a_{i,1} : v_{i,1}, \dots, a_{i,n_i} : v_{i,n_i}\}$ is defined as a set of paths:

$$s_i = \{p_1, \dots, p_{m_i}\}$$

Each p_j is a path derived from the document value. For multiple nesting levels, the path is extracted recursively to find the absolute navigational path starting from the root to the atomic value that can be found in the document hierarchy.

A schema s_i of document d_i is formally defined as follows:

$\forall j \in [1..n_i]$,

- if $v_{i,j}$ is atomic, $s_i = s_{d_i} \cup \{a_{i,j}\}$;
- if $v_{i,j}$ is an object, $s_i = s_i \cup \{a_{i,j}\} \cup \{\cup_{p \in s_{i,j}} a_{i,j}.p\}$ where $s_{i,j}$ is the schema of $v_{d_i,j}$;
- if $v_{i,j}$ is an array, $s_i = s_i \cup \{a_{i,j}\} \cup_{k=1}^{\|v_{i,j}\|} \Big(\{a_{i,j}.k\} \cup \{\cup_{p \in s_{i,j,k}} a_{i,j}.k.p\} \Big)$ where $s_{i,j,k}$ is the schema of the k^{th} value from the array $v_{i,j}$.

Example. Let us consider the collection $C = \{d_1, d_2, d_3, d_4, d_5\}$ composed of the documents introduced in section 1, figure 1. The underlying schema for the documents is described as follows:

```
s1 = { name, title, year }
s2 = { name, book, book.title, book.year }
s3 = { name, birthyear }
s4 = { name, artwork, artwork.1, artwork.2
     , artwork.1.title, network.1.year
     , artwork.2.title, artwork.2.year }
s5 = { name, title, year }
```

We can notice that the attribute *"book"* from document d_2 is an object in which are nested the attributes *"title"* and *"year"*. So, that leads to handling two different navigational paths *"book.title"* and *"book.year"*. We can also notice that the attribute *"artwork"* in document d_4 is an array which is composed of two sub-documents having the following sub-schemas:

```
s4.1 = { title, year }
s4.2 = { title, year }
```

Thus, that leads us to add to the dictionary the four aforementioned paths starting from *"artwork"*.

Definition 4 *(Collection Schema).* The schema S_C inferred from collection C is defined as follows:

$$S_C = \bigcup_{i=1}^{c} s_i$$

Definition 5 *(Dictionary).* The dictionary $dict_C$ of a collection C is defined by a set of pairs:

$$dict_C = \{(p_k,\ \triangle_k)\}$$

- $p_k \in S_C$;
- $\triangle_k = \{p_k\} \bigcup \{\bigcup_{\forall p_i \in S_C | p_i = p_l.p_k} p_i\}$. For each path p_k, $\triangle_k$ is the set of paths leading to p_k.

Example. The dictionary $dict_C$ constructed from the collection C of figure 1 is defined hereafter. Each dictionary entry p_k refers to the set of all extracted navigational paths $\triangle_k$. For example, the entry *"year"* refers to all navigational paths $\{year,\ book.year,\ artwork.1.year,\ artwork.2.year\}$ leading to the attribute "*year*".

```
{
    (name, {name}),
    (title, {title, book.title,
      artwork.1.title, artwork.2.title}),
    (year, {year, book.year,
      artwork.1.year, artwork.2.year}),
    (book, {book}),
    (book.title, {book.title}),
    (book.year, {book.year}),
    (birthyear, {birthyear}),
    (artwork, {artwork}),
    (artwork.1, {artwork.1}),
    (artwork.1.title, {artwork.1.title}),
    (artwork.1.year, {artwork.1.year}),
    (artwork.2, {artwork.2}),
    (artwork.2.title, {artwork.2.title}),
    (artwork.2.year, {artwork.2.year})
}
```

3.2 Querying Heterogeneous Document Stores

The querying process is supported by a set of elementary operators. These operators are expressed by native MongoDB query commands such as *"find"* or *"aggregate"*.

3.2.1 Kernel of Operators

The queries are defined according to combinations of elementary operators. The set of operators forms a kernel, which is denoted K. For now, this kernel is composed of three operators: projection, restriction (or selection) and aggregation. Each elementary operator is unary; we call C_{in} the queried collection, and C_{out} the resulting collection.

Definition 6 *(Kernel).* The kernel K is a minimal closed set composed of the following unary operators.

$$k = \{\pi,\ \sigma,\ \gamma\}$$

- $\pi_A(C_{in}) = C_{out}$ is a project operator, which consists in restricting each document schema s_i to a subset of attributes $A \subseteq S_{c_{in}}$.
- $\sigma_p(C_{in}) = C_{out}$ is a restrict operator, which consists in selecting documents from C_{in} satisfying the predicate p. A simple predicate is expressed by $a_k\ \omega_k\ v_k$ where $a_k \subseteq S_{C_{in}}$ is an *attribute*, $\omega_k \in \{=\ ;>\ ;<\ ;\neq\ ;\geq\ ;\leq\ \}$ is a comparison operator, and v_k is a value. It is possible to combine predicates by logical connectors $\{\ \vee,\ \wedge,\ \neg\}$. We suppose that the predicate is defined as, or normalized to, a conjunctive normal form:

$$\bigwedge_k \left(\bigvee_l a_{k,l}\ \omega_{k,l}\ v_{k,l} \right)$$

- ${}_G\gamma_F(C_{in}) = (C_{out})$ is an aggregate operator, which consists of aggregating each group of documents having same values for $G \subseteq S_{C_{in}}$ and calculating the aggregate values, $F = \{f(a_k) | f \in \{Sum, Max, Min, Avg, Count\} \wedge a_k \in S_{C_{in}} \wedge a_k \notin G\}$.

Definition 7 *(Query).* A query Q is formulated by composing previous unary operators as follows:

$$Q = q_1 \circ \cdots \circ q_r(C)$$

where $\forall i \in [1,r],\ q_i \in K$.

Example. Let us consider the collection C of figure 1. We propose hereafter several examples of queries; let us staying aware that structural heterogeneity exists in C and that those queries are not expected to deal with the heterogeneity.

- "Search for the list of authors' name and their publications"
  $\pi_{name,title}(C) =$

```
[ { name:"victor hugo",
    title:"les miserables"    },
  { name:"honore de balzac"   },
  { name:"paul verlaine"      },
  { name:"charles baudelaire" },
  { name:"pierre de ronsard",
    title:"les amours"        } ]
```

- "'Search for the titles of the publications of *Pierre de Ronsard* and *Charles Baudelaire*"
  $\pi_{(name,title}(\sigma_{name=\text{"Charles Baudelaire"} \vee name=\text{"Pierre de Ronsard"}}(C)) =$

```
[   { name:"charles baudelaire" },
    { name:"pierre de ronsard",
        title:"les amours" } ]
```

- "'Search for the number of publications for each authors"
  ${}_{name}\gamma_{count(title)}(C) =$

```
[ { name:"victor hugo", count:1 },
  { name:"honore de balzac", count:0 },
  { name:"paul verlaine", count:0 },
  { name:"charles baudelaire", count:0 },
  { name:"pierre de ronsard", count:1 } ]
```

As aforementioned, due to the structural heterogeneity of the attribute *"title"* we notice that these queries do not give relevant results according to the stored documents. To obtain relevant results users would have to write complex queries taking into account the various schemas.

3.2.2 Query Extension Process

Dealing with a collection of heterogeneous documents complicates the process of expressing queries. Most of NoSQL systems do not give native support to query heterogeneous documents. For instance, the *"find"* operator, as well as the *"aggregate"* pipeline operator of MongoDB, is not able to automatically recognize the numerous structures of the queried collection. More precisely, the result does not include values from navigational paths that are not explicitly included in the query.

Our approach aims at enabling a transparent querying process on a collection of heterogeneous documents via an automatic query rewriting process. It employs the materialized dictionary to enrich the initial query by including the different navigational paths that lead to desired attributes. It is described in the algorithm 1 and parts are described hereafter:

- In case of projection, the list of projected attributes A is extended by the various navigational paths $\triangle_k$ for each attribute $a_k \in A$; the underlying idea is to ask the dbms to search for all possible existing path for attributes.
- In case of restriction, the normal conjunctive form of the predicate p is enriched by the set of extended disjunctions built from the navigational paths $\triangle_{k,l}$ for each attribute a_j of the predicate; the underlying idea is to ask the dbms to test all possible existing paths for attributes.
- In case of aggregation, the operation is extended using two operations: an added projection to deal with the heterogeneity of attributes, and a classical aggregation to operate the calculus. The list of attributes G is extended by the various navigational paths $\triangle_k$ for each attribute $a_j \in G$. Each path is renamed according to the attribute a_j given in the aggregation; in the algorithm 1, we note the rename operation $a_j \Leftarrow \triangle_j$. An equivalent projection is made for all attributes of F. Then the true and classical aggregation can be done. The underlying idea is to ask the dbms to "flatten" all possible heterogeneous paths of attributes in G and F in order to be able to group documents on the same value of a same attribute and calculate the aggregated value on a same attribute. Let us notice that such operations are done by the dbms, often in pipeline mode, and is not a physical factorization (nor a physical flattening).

Example. Let us consider the previous queries examples, section 3.2.1.

- The query rewriting engine rewrites the query $\pi_{name,title}(C)$ and for each projected field (respectively *"name"* and *"title"*), the process consults the dictionary and extracts all the possible navigational paths (respectively $\triangle_{name} = \{name\}$, and $\triangle_{title} = \{title, book.title, artwork.1.title, artwork.2.title\}$). The projection query is then rewritten as $\pi_{name,title,book.title,artwork.1.title,artwork.2.title}(C) =$

```
[ { name:"victor hugo",
    title:"les miserables" },
  { name:"honore de balzac",
```

Algorithm 1: Automatic extension of the initial user query.

```
input: $Q$
output: $Q_{ext}$
$Q_{ext} \leftarrow id$                                  // identity
foreach $q_i \in Q$ do
  switch $q_i$ do
    case $\pi_{A_i}$                                     // projection
    do
      $A_{ext} \leftarrow \bigcup_{\forall a_k \in A_i} \triangle_k$
      $Q_{ext} \leftarrow Q_{ext} \circ \pi_{A_{ext}}$
    end
    case $\sigma_{Norm_p}$                               // restriction
    do
      $P_{ext} \leftarrow \bigwedge_k (\bigvee_l \bigvee_{a_j \in \triangle_{k,l}} a_j \; \omega_{k,l} \; v_{k,l})$
      $Q_{ext} \leftarrow Q_{ext} \circ \sigma_{P_{ext}}$
    end
    case ${}_G\gamma_F$                                  // aggregation
    do
      $Q_{ext} \leftarrow Q_{ext} \circ (\pi_{\bigcup_{\forall a_j \in G} a_j \Leftarrow (\triangle_j), \bigcup_{\forall f_k(a_k) \in F} a_k \Leftarrow (\triangle_k)} \circ {}_G\gamma_F)$
    end
  end
end
```

```
    book:{title:"le pere Goriot" } }
  { name:"paul verlaine" },
  { name:"charles baudelaire",
    artwork:[
      {title:"les fleurs du mal"},
      {title:"le spleen de Paris"}] },
  { name:"pierre de ronsard",
    title:"les amours" } ]
```

- Our rewriting engine extends the query $\pi_{name,title}(\sigma_{name=\text{“Charles Baudelaire”} \vee name=\text{“Pierre de Ronsard”}}(C))$ with the dictionary entries in the same way as the previous query. The projected attributes are extended as for the previous query. Next, the process continues with the selection query. The selection is rewritten by extending the normal form of its predicate; the attribute *“name”* has only one structural form, then the predicate is not rewritten. The composed query is then rewritten as $\pi_{name,title,book.title,artwork.1.title,artwork.2.title}$ $(\sigma_{name=\text{“Charles Baudelaire”} \vee name=\text{“Pierre de Ronsard”}})$ $(C)) =$

```
[ { name:"charles baudelaire",
    artwork:[
      {title:"les fleurs du mal"},
      {title:"le spleen de Paris"}] },
  { name:"pierre de ronsard",
    title:"les amours" } ]
```

- Our rewritten query transforms the query ${}_{name}\gamma_{count(title)}(C)$ to a project operator introduced to rename the different heterogeneous paths. Then, the query is rewritten as a composed query such as ${}_{name}\gamma_{count(title)}(\pi_{name:(name \Leftarrow (name)), title:(title \Leftarrow (title|book.title|artwork.1.title|artwork.2.title))}$ $(C)) =$

```
[ { name:"victor hugo", count:1 },
  { name:"honore de balzac", count:1 },
  { name:"paul verlaine", count:0 },
  { name:"charles baudelaire", count:2 },
  { name:"pierre de ronsard", count:1 } ]
```

4 EXPERIMENTS

The overall goal of the next experiments is to study if the rewriting process is acceptable along many dimensions: cost/overhead for query evaluation, size of the dictionary and cost time for building it, number of possible schemas that *EasyQ* can deal with. The purpose of our first experiments in this section is to study the scale effects on the rewritten queries regarding two main factors: the size of the queried collection and the heterogeneity levels. In addition, we study their effects on the dictionary. We choose MongoDB to store the different datasets, the dictionary and to run the rewritten queries.

Let us notice that, using MongoDB or any other classical document store, the rewriting process is compatible with the underlying dbms engine. Indeed, during any query evaluation, if a path is not present in a document, it is simply ignored. Thus, the following rules are applied during queries evaluation:

- Projection: for each document, any non-present projection path is ignored and only those really existing in the document are retrieved.
- Restriction: for each document, any non-present enriched path is ignored since it has been included in a disjunctive form. If no path is found in the document, the condition is evaluated to false.
- Aggregation: The same rule applies than for projection since we use this operator for rewriting purpose; grouping and aggregation computing are classical ones.

Experimental Protocol. All experiments in this paper were implemented in Python and ran on a server with Intel I5 (3.4 GHz 4 cores), 16GB RAM and CentOS 7.0. We repeated each experiment 5 times and we report the mean values. The details of the dataset and the queries are presented in the remainder of this section.

Dataset. In this experimental evaluation, we employ synthetic datasets with various schemas and volume. All datasets are generated from the initial flat collection of documents that describe films published by IMDB[1]. To this end, we inject the structural heterogeneity by introducing new grouping fields. We nest the initial attributes inside these new groups. The values of those fields are randomly chosen from the original film collection. To add more complexity, we can set the nesting level used for each generated structure. We built our custom data generator allowing us to define several parameters such as the number of schemas to produce in the collection, the percentage of the presence of every generated schema. For each schema, we can adjust the number of grouping objects. We mean by grouping object, a compound field in which we nest a subset of the document.

Let us notice that every dataset can be generated in two versions: the generated heterogeneous one and an equivalent flat one. The flat dataset contains data from the heterogeneous one in which each document is flatten to its leaf attributes. So, each document values existing in heterogeneous dataset also exists in the flat one. This allows to compare queries over heterogeneous data and equivalent homogeneous data (flat documents in our case) since, if they are relevant, they should return the same number of documents and the same values. We are currently working on the online free delivery of datasets and datasets generator; for the moment you can ask the authors or visit their websites.

Queries. We define two queries composed of 2 and 8 predicates for projection and selection operators, and we use all possible comparison operators on different data types. We generate for each query two versions constituted by the conjunctive form of the predicates in Q_1, Q_3 queries, and disjunctive in Q_2, Q_4 queries. Moreover, we introduce two other queries to study the aggregation operator A_1, A_3.

- Q_1 & Q_2 select all documents where the *"director name"* of the film starts with the letter *"A"* and/or the film got as *"gross"* more than *100 K*.
- Q_3 & Q_4 select all documents where the *"director name"* of the film starts with the letter *"A"* and/or the film got as *"gross"* more than *100 K* and/or the *"duration"* of the film does not exceeds *200 minutes* and/or the *"title year"* is less than the year *1950* and/or the *"country"* of the film is known and/or the film *"language"* is *"English"* and/or the film got *"IMDB score"* more less than *4* and/or the *"number of Facebook likes"* is greater than *500*.
- A_1 group documents by *"country"* and *"language"* and then aggregate by the function *"Max"* over the *"film score"*.
- A_2 group documents by *"director name"* and *"year"* and then aggregate by the function *"Sum"* over the *"revenue"*.

[1]www.omdbapi.com

Table 1: Settings of the generated dataset for rewritten queries evaluation.

Setting	Value
# schemas	10
# groups per schema	{5,6,1,3,4,2,7,2,1,3}
Nesting levels per schema	{4,2,6,1,5,7,2,8,3,4}
% schema presence	10%
#attributes per schema	Random
#attributes per group	Random

Scale Effects on the Rewritten Queries. In this test series, we try to study the effects of the scale on the rewritten queries. We define three contexts in which we run the above-defined queries. The order of query execution is set to be random to prevent the document store from reusing cache mechanisms. Here, we describe the different execution contexts:

- We note *"QBase"* the query that refers to the initial user query (one of the above defined queries), and that is executed over the homogeneous version of the dataset. The purpose of this first context is to study the native behaviour of the document stores. We use this first context as a baseline for our experimentation.
- The *"QRewritten"* refers to the query *"QBase"* rewritten by our approach and executed over the heterogeneous version of the datasets. As aforementioned the two datasets are considered "equivalent", then *"QRewritten"* is expected to return the same number of documents (and content) than *"QBase"*. It is the case in all the following experiments.
- The *"QAccumulated"* refers to the set of equivalent queries formulated on each possible schema from the collection. In our case, it is made of 10 separated queries since we are dealing with collections having ten schemas. It is executed over the heterogeneous version of the datasets. For the experiments, we wrote these queries "by hand".

Table 1 presents the characteristics of the datasets used for this first category of experiments. Let us notice that each attribute is present in ten different schemas at different nesting levels.

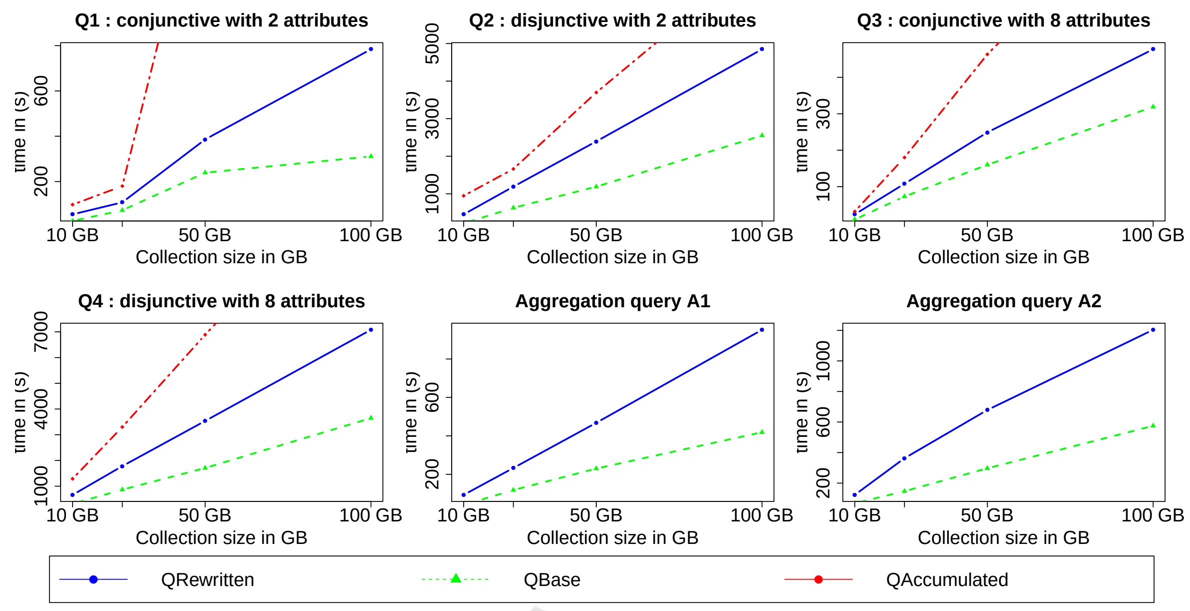

Figure 3: Collection size effect on the rewritten queries compared to classical ones.

Figure 3 shows our first results. Each graphic shows: x axis is the collection size (GB), y axis is the time of query execution (s), blue curve refers to *"QRewritten"*, green one to *"QBase"*, and red one to *"QAccumulated"*, that is the sum of the evaluation of the ten sub-queries.

As shown in figure 3, the behaviour of our rewritten query is similar to the baseline. Both *"QRewritten"* and *"QBase"* have execution time linear when regarding collection size while the accumulated query *"QAccumulated"* seems to exhibit exponential time costs. We can notice also that the execution of our solution is less than two times higher (e.g., disjunctive form) than the normal execution of the baseline query. Moreover, we score an overall overhead that does not exceed 1,5 times in the different projection and selection queries.

The same behaviour is also noticed while studying the aggregation queries. Only *"QRewritten"* and *"QBase"* are presented in the study of the aggregation queries. The rewriting of aggregation uses the *"aggregate"* pipeline operator of MongoDB. It is remarkable that although the necessary insertion of two projections in the pipeline (cf. algorithm 1 and explanations section 3.2.2), the time execution overhead remains low.

For all queries, despite of the fact that each attribute has been replaced by ten possible paths, the time execution overhead remains quite low. We believe that this overhead is acceptable since we bypass the extra costs for refactoring the underlying data structures. Unlike the baseline, our synthetic dataset contains different grouping objects with varying nesting levels. Then, the rewritten query contains several navigational paths that are processed by the native query engine of MongoDB to find matches in each visited document among the collection. Finally, let us notice that the aggregation rewriting allows performing complex computations that are particularly time-consuming and prone to errors when done "by hand".

Heterogeneity Effects on the Dictionary and the Query Build Time. With this series of experiments, we try to push the dictionary and the query rewriting engine to their limits. For that, we generated a heterogeneous synthetic collection of 1 GB. We use the initial 28 attributes from the IMDB flat films collection. The custom collections are generated in a way that each schema inside a document is composed of two grouping objects with no further nesting levels. We generated collection having *10, 100, 1k, 3k* and *5k* schemas. For this experiment, we test the use of the query Q_4 introduced earlier in this section. We present the dictionary size and the time needed to build the rewritten query of Q_4 in Table 2

It is notable that the time to build the rewritten query is very low, always less than two seconds when 5K distinct schemas exist in the collection. In addition, it is possible to construct a dictionary over a highly heterogeneous collection of documents, here

Table 2: Data diversity effects on query rewriting time and dictionary size.

# of schemas	Query rewriting in (s)	Dictionary size
10	0.0005	40 KB
100	0.0025	74 KB
1 K	0.139	2 MB
3 K	0.6	7.2 MB
5 K	1.52	12 MB

our dictionary can support up to 5k of distinct schemas. The resulting size of the materialized dictionary is very promoting since it does not require significant storage space. Furthermore, we also believe that the time spent to build the rewritten query is very interesting and represent another advantage of our solution. When rewriting the queries, we try to find distinct navigational paths for eight predicates. Having 5k of paths for each query predicate, these experiments show that we are able to generate a selection query with 40k of navigational paths expressed in disjunctive form.

5 CONCLUSION

In this paper, we provide a novel approach for querying heterogeneous documents describing a given entity over document-oriented data stores. Our objective is to allow users to perform their queries using a minimal knowledge about data schemas. Our tool *EasyQ* is based on two main principles. The first one is a dictionary that contains all possible paths for a given field. The second one is a rewriting module that modifies the user query to match all field paths existing in the dictionary. Our approach is a syntactic manipulation of queries. Therefore, it is grounded on a strong assumption: the collection describes homogeneous entities, i.e., a field has the same meaning in all document schemas. If this assumption is not guaranteed, users may face with irrelevant or incoherent results.

We conduct experiments to compare the execution time cost of basic MongoDB queries and rewritten queries proposed by our approach. We conduct a set of experiments by changing two primary parameters, the size of the dataset and the structural heterogeneity inside a collection. Results show that the cost of executing rewritten queries proposed in this paper is higher when compared to the execution of basic user queries. The overhead added to the performance of our query is due to the combination of multiple access path to a queried field. Nevertheless, this time overhead is neglectful when compared to the execution of separated queries for each path. Let us notice that an interesting advantage of *EasyQ* is that each time a query is evaluated, it is first rewritten according to the dictionary taht is updated online. Therefore, the query will always automatically deal with all existing schemas.

These first results are very encouraging to continue this research way and need to be strengthened. Short-term perspectives are to continue evaluations and to identify the limitation regarding the number of paths and fields in the same query and regarding time cost. More experiments still to be performed on larger ”real data” datasets. Another perspective is to study in depth the process of the dictionary building in real applications and in parallel of collection updates and querying.

Finally, a long-term perspective is to enhance querying over a collection of documents presenting several levels of heterogeneity, i.e., structural as well as syntactic and semantic heterogeneities.

REFERENCES

Baazizi, M.-A., Lahmar, H. B., Colazzo, D., Ghelli, G., and Sartiani, C. (2017). Schema inference for massive json datasets. In *(EDBT)*.

Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., Siméon, J., and Stefanescu, M. (2002). Xquery 1.0: An xml query language.

Bourhis, P., Reutter, J. L., Suárez, F., and Vrgoč, D. (2017). Json: data model, query languages and schema specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 123–135. ACM.

Chasseur, C., Li, Y., and Patel, J. M. (2013). Enabling json document stores in relational systems. In *WebDB*, volume 13, pages 14–15.

Clark, J., DeRose, S., et al. (1999). Xml path language (xpath) version 1.0.

DiScala, M. and Abadi, D. J. (2016). Automatic generation of normalized relational schemas from nested key-value data. In *Proceedings of the 2016 International Conference on Management of Data*, pages 295–310. ACM.

Florescu, D. and Fourny, G. (2013). Jsoniq: The history of a query language. *IEEE internet computing*, 17(5):86–90.

Hai, R., Geisler, S., and Quix, C. (2016). Constance: An intelligent data lake system. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2097–2100. ACM.

Herrero, V., Abelló, A., and Romero, O. (2016). Nosql design for analytical workloads: variability matters. In *ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings 35*, pages 50–64. Springer.

Lin, C., Wang, J., and Rong, C. (2017). Towards heterogeneous keyword search. In *Proceedings of the ACM Turing 50th Celebration Conference-China*, page 46. ACM.

Papakonstantinou, Y. and Vassalos, V. (1999). Query rewriting for semistructured data. In *ACM SIGMOD Record*, volume 28, pages 455–466. ACM.

Rahm, E. and Bernstein, P. A. (2001). A survey of approaches to automatic schema matching. *the VLDB Journal*, 10(4):334–350.

Ruiz, D. S., Morales, S. F., and Molina, J. G. (2015). Inferring versioned schemas from nosql databases and its applications. In *International Conference on Conceptual Modeling*, pages 467–480. Springer.

Sheth, A. P. and Larson, J. A. (1990). Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys (CSUR)*, 22(3):183–236.

Shvaiko, P. and Euzenat, J. (2005). A survey of schema-based matching approaches. *Journal on data semantics IV*, pages 146–171.

Tahara, D., Diamond, T., and Abadi, D. J. (2014). Sinew: a sql system for multi-structured data. In *Proceedings of the 2014 ACM SIGMOD*, pages 815–826. ACM.

Wang, L., Zhang, S., Shi, J., Jiao, L., and Hassanzadeh (2015). Schema management for document stores. *Proceedings of the VLDB Endowment*, 8(9):922–933.

Yang, Y., Sun, Y., Tang, J., Ma, B., and Li, J. (2015). Entity matching across heterogeneous sources. In *Proceedings of the 21th ACM SIGKDD*, pages 1395–1404. ACM.