Towards schema-independent querying on document data stores

Hamdi Ben Hamadou
Université de Toulouse, UT3, IRIT, (CNRS/UMR 5505)
Toulouse, France
hamdi.ben-hamadou@irit.fr

Faiza Ghozzi
Université de Sfax, ISIMS, MIRACL
Sfax, Tunisia
faiza.ghozzi@isims.usf.tn

André Péninou
Université de Toulouse, UT2J, IRIT, (CNRS/UMR 5505)
Toulouse, France
andre.peninou@irit.fr

Olivier Teste
Université de Toulouse, UT2J, IRIT, (CNRS/UMR 5505)
Toulouse, France
olivier.teste@irit.fr

ABSTRACT

Document is a pervasive semi-structured data model in today's Web and the Internet of Things (IoT) applications where the data structure is rapidly evolving over time. NoSQL document-oriented databases are well-tailored to efficiently load and manage massive collections of heterogeneous documents without any prior structural validations. However, this flexibility becomes a serious challenge while querying a heterogeneous collection of documents. Hence, it is mandatory for users to reformulate original query or to formulate new ones when more structures arrive in the collection. In this paper, we propose a novel approach to build schema-independent queries designed for querying multi-structured documents. We introduce a query enrichment mechanism that consults a pre-materialized dictionary defining all possible underlying document structures. We automate the process of query enrichment via an algorithm that rewrites select and project operators to support multi-structured documents. To study the performances of our proposed approach we conduct experiments on synthetic dataset. First results are promising when compared to the normal execution of queries on homogeneous dataset.

1 INTRODUCTION

The popularity of NoSQL systems is growing in the database community thanks to their ability to store and query schema-free data in flexible and efficient ways [8, 21]. The document data model is pervasive in the most Web and the Internet of Things (IoT) applications [13], and several database systems support this data model in an efficient way [1, 4, 5]. Furthermore, in such applications, the structures of documents representing same entity are subject to structural changes [7]. An application may face the problem of dealing with multi-structured data [2]. To formulate relevant queries, there is a need to have a precise knowledge of data structures because document stores do not provide native support for querying multi-structured data. Thus, it is mandatory to manually include all possible navigational paths for the attributes of interest to formulate relevant query. The structural changes require users to reformulate original query which is a time-consuming and prone to error task. The challenge addressed in this paper is how to support querying upon future structural heterogeneity without affecting the application code.

In the context of document-oriented databases and due to the flexible nature of documents, it is possible to create a collection of documents describing a single entity with multiple structures. This characteristic points to several kinds of heterogeneity [20]. The structural heterogeneity refers to diverse representation of documents, (e.g.: nested or flat structures, nesting levels, etc.) as shown in Figure 2. The syntactic heterogeneity is a result of differences in representation of data, (e.g. "*movie_title*" or "*movieTitle*"). Moreover, the semantic heterogeneity is presented when the same fields may rely on distinct concepts in separate documents. The aim of this paper is to focus on structural heterogeneity.

The problem of schema-independent querying is a hot topic in the study of document-oriented databases for both industry and academia [6, 22, 23]. Previous work from the literature resolved this issue with the following two approaches: (i) performs physical integration of data by mapping integrated document structures into a unified structure [22] and (ii) performs a virtual integration by introducing a custom interface that proposes new virtual schema to be learned by the users while querying heterogeneous data [23]. The first approach modifies the underlying data structure, which is not possible while supporting legacy applications designed to run over original data structures. Moreover, this approach implies to define the mapping for any original data structure. The second one requires more efforts from the user to learn new global structures. This approach is a time-consuming task and possibly prone to be error when there is a need to query new documents with new structures since all queries are subject to revisions.

In this paper, we propose a novel approach to build schema-independent queries designed for querying multi-structured documents. We propose a virtual integration that runs in a transparent way, hides the complexity to build expected queries, and supports structural heterogeneity evolution. Always, we rewrite the queries during the execution time to guarantee the usage of the latest structures of documents as defined in the dictionary.

The problem of structural heterogeneity refers to the possibility to find different navigational paths that lead to the same attribute. The attributes are not located at the same position inside documents, and having a limited knowledge of navigational paths is insufficient to retrieve the required information. In Figure 1, the attribute "*country*" in documents describing films may not be relevant to differentiate between "*actor.country*" or "*director.country*." Some sub-paths may help to resolve the ambiguity such as "*actors.country*" and "*director.country*" anywhere in the document. Therefore, some sub-paths may be used rather than attributes names. In all cases, we hypothesise that there exist some navigational paths to differentiate the different entities contained in the document.

```
{
    "movie_title":"Fast and Furious",
    "country": "USA",
    "actors": [ { "name": "Vin Diesel",
                  "country": "USA"
                }, ... ],
    "director" : { "name": "F. Gray Gray",
                   "country": "USA"
                 }
}
```

Figure 1: Descriptive fields ambiguity

We introduce the *EasyQ*, that stands for "*Easy Query*," as a tool to validate our approach. We give particular interest to MongoDB for the implementation and evaluation. The primary contribution of our work is to reformulate users original queries formulated based on simple knowledge of the descriptive field or sub-paths that contains the desired information. The users' queries are formulated based on a schema-independent fashion, i.e. users can formulate an initial query based on a subset of possible schemas without carrying about all available structures. Query rewriting engine is responsible to transparently reformulate the initial query to match with all existing schemas returning a relevant result. To deal with document schema heterogeneity, we define a dictionary that contains all possible paths for all existing fields. The query rewriting engine enriches the user query with all possible paths found in the dictionary for each field used in the user query. In this paper, we use interchangeably the terms field, descriptive field and attribute.

The rest of our paper is structured as follows: In section 2, we illustrate the paper issue. Section 3 reviews the most relevant works, providing support to query multi-structured documents. Section 4 describes in details our approach. Section 5 presents our first experiments and the performances of our approach while changing the size and the number of schemas per collection. In Section 6, we summarize our findings.

2 QUERYING DOCUMENT STORES WITH MULTIPLE SCHEMA ISSUES

As discussed earlier, querying multi-structured data is a complex task. The problem is which structure to use while formulating queries and how this choice affects the results. In the following, we present a simple illustrating example.

Let $C = \{d_1,\ d_2,\ d_3,\ d_4\}$ be a set of four films as documented in Figure 2. In this example we represent documents using JSON (JavaScript Object Notation). Most of the NoSQL systems support this notation of representing semi-structured data. A document d_i is defined by a key-value pair (i, v_i) where i is the key and v_i is the value described with JSON.

Let us consider that we want to retrieve information related to available languages for each presented movie. We formulate a projection query with the fields "*movie_title*" and "*language*." using MongoDB syntax as follows:

$$db.C.find(\{\}, \{\text{“}movie\_title\text{”} : 1, \text{“}language\text{”} : 1\}).$$

In this query, the field "*movie_title*" does not cause any difficulty since it is always at the same structural level in the four documents.Therefore, the query engine is able to locate all information related to the field "*movie_title*." However, the field "*language*" may cause some information loss since it is founded

```
d1: {
        "movie_title":"Fast and furious",
        "year":2017,
        "language":"English"
    },
d2: {
      "movie_title": "Titanic",
      "details":
        {"year":1997,"language":"English"}
    },
d3: {
        "movie_title": "Despicable Me 3",
        "year":2017
    },
d4: {
        "movie_title": "The Hobbit",
        "versions":
        [{"year":2012, "language":"English"},
        {"year":2013, "language":"French"}]
    }
```

Figure 2: Four documents of films collection

at several positions across documents. Thus, assuming that we formulate a query with limited knowledge of the structure s_2 from d_2, we build a query with the fields "*movie_title*" and "*details.language*."

$$db.C.find(\{\}, \{\text{“}movie\_title\text{”} : 1, \text{“}details.language\text{”} : 1\})$$

Executing such query in MongoDB leads to an incomplete result since "*details.language*" field is not available in documents $d_1,\ d_3,\ and\ d_4$. The problem comes from the structural heterogeneity due to the different structural position of the field "*language*," i.e. "*language*" in document d_1, "*details.language*" in document d_2, and "*versions.language*" in document d_4. Hence, we may include all these paths in the query using specific and often complex syntax.

Moreover, we can try to formulate two different queries. The first one is formulated over schema s_1 of the document d_1 in order to retrieve the list of titles and "*language*" for each film. We use the following MongoDB query:

$db.C.find(\{\}, \{\text{“}movie\_title\text{”} :\ 1, \text{“}language\text{”} :\ 1\})$

We than get the following result:

C_1 = [
{“*movie_title*” : “*Fast and furious*”, “*language*” : *English*},
{“*movie_title*” : “*Titanic*”},
{“*movie_title*” : “*Despicable Me* 3”},
{“*movie_title*” : “*The Hobbit*”}]

We formulate the second using the schema s_4 of document d_4:

$db.C.find(\{\}, \{\text{”}movie\_title\text{”} :\ 1, \text{”}versions.language\text{”} :\ 1\})$

We get the following result:

C_2 = [
{“*movie_title*” : “*Fast and furious*”},
{“*movie_title*” : “*Titanic*”},
{“*movie_title*” : “*Despicable Me* 3”},
{“*movie_title*” : “*The Hobbit*”, “*versions*” :
[{“*language*” : “*English*”}, {“*language*” : “*French*”}]
]

When executing both queries, the query engine returns two results. As expected, all possible information related to the field "*movie_title*" is returned for all documents as it is located on document's root. For the first query, only first document matches with the field containing "*language*" information. The second query succeeded to retrieve "*language*" information only from the fourth document. The challenge is how to formulate a single query and retrieve all information related to the field "*language*" without any redundancy. For instance, the same information related to the field "*movie_title*" is obtained twice in the resulted collections C_1 & C_2.

To solve this we introduce a transparent way to build relevant schema-independent queries that bypass structural heterogeneity in documents stores. A simple knowledge of the required attributes allows users to retrieve adequate documents regardless the structural heterogeneity in the collection. This ease simplifies the task for end-users and provides them an efficient way to retrieve information of interest. In case of there-above example, we enrich the original user query by adding all possible navigational paths to retrieve relevant documents. For instance, we formulate the query $db.C.find(\{\}, \{\text{“}movie\_title\text{”} : 1, \text{“}language\text{”} : 1, \text{“}details.language\text{”} : 1, \text{“}versions.language\text{”}\})$ in MongoDB syntax. It can bypass the structural heterogeneity in the current state of the collection and it projects all desired values.

3 STATE OF THE ART

The widespread use of semi-structured data gives increased interest to build solutions enabling queries over semi-structured data. We distinguish existing solutions systems on the basis of the proposed querying approach: 1) schema-dependent querying approach that requires knowledge of the schema in a similar way as conventional relational database systems, 2) schema-independent querying approach that does not need any prior schema knowledge from the user and is able to extract the schemas at querying time.

The first category of systems is designed to enable queries based on reliable knowledge about the schema or the navigational paths for desired values when dealing with nested data. Such systems offer complicated querying language such as regular expressions with XQuery or Xpath [17] when dealing with XML data. XQuery works with the structure to retrieve precisely the desired results. However, if the user does not know the structure, it is impossible to write the relevant query. Moreover, a single query is generally not able to retrieve data when several schemas are to be considered simultaneously. We can notice the same considerations with JSONiq [9], the extension of XQuery, designed to deal with large-scale data such as JSON data. Other systems suggest JavaScript queries API, the case of MongoDB [5], to build a query by specifying a document with properties expected to match with the results. It offers a broad range of querying capabilities, in particular data processing pipelines. The API requires a complex syntax and it is necessary that queries explicitly include all the various schema structures within documents to access data. Otherwise, the query engine returns only documents that match the supplied criteria even if the fields with the desired information exist but under other paths than those existing in the query. Another kind of works is SQL++ [19] relies on the rich SQL querying interface. In this case, it is also mandatory to express all exact navigational paths in order to obtain the desired results.

The above-studied systems are designed to support queries over semi-structured data with known schemas. To formulate queries user needs to know the exact underlying data structures. Also, they neglect the fact that user may have limited knowledge about the data structure and hence may be unable to formulate correct queries over the heterogeneous dataset.

To overcome these limitations, recent works were conducted to enable schema-independent querying; the second category underlined at the beginning of this section. Thus, the schema is not mandatory to be known in advance at loading time. We classify the studied works according to two approaches: (i) performs physical integration by refactoring integrated data structures into an unified structure; and (ii) adopts virtual integration by introducing either a custom interface and/or a new query language [23].

In the first direction, several works were designed to deal with semi-structured data. Those works share the idea of the schema-on-read. There is no need to define schemas before loading data, they infer the implicit schema later from stored datasets on query time. They expose for the users a relational view over the data to help them to build SQL queries. Sinew[22], is able to infer schemas from semi-structured data. It defines for the user a logical view on the inferred schema, and it flattens data into columns to be stored into relational database system (RDBMS). Drill[12] enables schema-independent querying via SQL over heterogeneous data without first defining a schema. It gives support for nested data. Tenzing[18] infers a relational schema from the underlying data but can only do so for flat structures that can be trivially mapped to a relational schema.

The principle of the previous solutions suggests heavy physical refactorization that requires flattening the underlying data structures into a relational format using complex encoding techniques. Hence, the refactorization requires additional resources such as the need for external relational database and extra efforts to learn the unified inferred relational schema. Besides, some solutions do not support the flexible nature of semi-structured data [18] for instance they cannot handle nested data. User dealing with those systems has to learn new schemas every time the workload changes, or new data comes because there is a need to re-generate the relational view and the stored columns after every change.

Virtual integration gets also attention from researchers [14, 23]. Works are inspired by the data lake approach [11] where data is collected in their original format for later use. We consider two major classes: i) schema-oriented queries; and ii) keyword querying.

Works from the first class infer the schema from a collection of data and offer for the users the possibility to query the inferred schema and to check whether a field or sub-schema exists or not to guide them while developing their applications. In [23] the authors propose to summarize all the document schema under a skeleton to discover the existence of fields or sub-schemas inside the collection. In [14] the authors suggest extracting all the schema that are present in the collection to help final users to be aware of the schemas and all fields in the integrated collection of documents. These solutions are limited only to type and field identification and are not used to determine the different paths to access a field in the collection.

Keyword querying has been adopted in the context of XML [10, 24]. The process of answering a keyword query on XML data starts by identifying the existence of the keywords within the documents (possibly through some free-text search). They

take as input the searched keywords and return a subset from the document that matches with the query keywords. A score is computed based on the structure of sub-documents, and according to this score, the respective XML documents containing all the keywords are returned.

Works in Keyword querying suggest doing a pairwise comparison or binary search to identify the possible positions for queried keywords. This concept is not well tailored for a large number of documents with complex structures (different nested elements, numerous attributes, etc.).

From state of the art, we build our approach in the idea of offering virtual integration to enable schema-independent querying via the usage of keyword based on the attributes and to support native semi-structured features such as nested attributes and support for heterogeneous collections of documents.

Our work relates in some way to previous attempts with XML keyword querying[16]. The most important contribution of these earlier efforts is to prevent users from learning complex underlying schemas as well as a complex query language to manage paths. We adopt this idea that the user may not be aware of all existing schemas and cannot manage too complex queries in order to enable schema-independent querying based on only knowledge about the field with the desired information. The main difference between our works and the keywords querying is that we require from the user some simple details about the queried data. For instance, if we execute a keyword query "*English*". It is possible to have as result "*language*" : "*English*" and also "*movie_title*" : "*Johnny English.*" With our approach, we will specify that we give interest to the field "*language*" = "*English*" or to the field "*movie_title*" contains "*English.*"

4 QUERYING HETEROGENEOUS COLLECTION OF DOCUMENTS

In our proposal, we want to enable queries over multi-structured documents by automatically handling the underlying structural heterogeneity. Thus, our query rewriting engine will give transparent support for the heterogeneity on both stored and future new data.

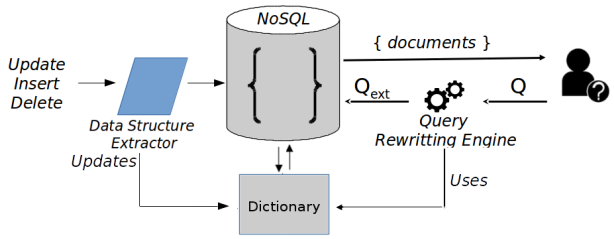

Figure 3: An overview of EasyQ

To give an overview of our approach, let us consider the following selection query (selection operation is defined later in this section):

$$\sigma_{(\text{“}year\text{”}=1997)}(C)$$

We refer to the collection presented in Figure 2 in which we notice that it exists three distinct navigational paths leading to the attribute "*year*," i.e. "*details.year*" "*versions.year*" and "*year*." For each document, at least one path can lead to the attribute "*year*." It is possible to express the selection predicate in disjunctive form of navigational paths.

$$X = \Big((\text{“}year\text{”} = 1997) \vee (\text{“}details.year\text{”} = 1997) \vee (\text{“}versions.year\text{”} = 1997) \Big)$$

We rewrite the initial query into $\sigma_{(X)}(C)$. Two conditions have to be satisfied to select one document, (i) it does exist at least one navigational path from the sub-conditions of X inside the document and (ii) the result of evaluating at least one of these sub-conditions is equal to true. Otherwise, X is equal to false and the document is not returned in the result.

The challenges are how to enable schema-independent querying in transparent ways and how to support future new structures without revising the application code.

Figure 3 gives a high-level illustration of our query rewriting engine called **EasyQ**. *EasyQ* is designed to be used early in data loading phase to materialize a dictionary that tracks the different navigational paths, for all attributes. *EasyQ* is also used at querying time to enrich the query Q of the user and to bypass the structural heterogeneity. It takes as input the user query formulated over final fields or sub-paths, and the desired collection. The query rewriting engine produces one extended query Q_{ext} that will be executed by the underlying document store system. The result of this extended query is a collection containing relevant information. An important result of such architecture is that the same user query, evaluated at different moment, will be rewritten each time. So, if new documents with new structures have been inserted in the collection (or existing documents are updated), these new structures are automatically handled and results remain relevant with the query.

In the rest of this section, we describe the formal model of multi-structured documents, dictionary, and the querying operators across multi-structured documents. Finally, we formally define how we rewrite the queries.

4.1 Multi-structured data modeling

Definition 4.1 (Collection). A collection C is defined as a set of documents

$$C = \{d_1, \ldots, d_{|C|}\}$$

Each document d_i is considered as a *(key-value)* pair where the value takes the form: $v_i = \{a_{i,1} : v_{i,1}, \ldots, a_{i,n} : v_{i,n}\}$

Definition 4.2 (Document). A document d_i, $\forall i \in [1, c]$, is defined as a *(key,value)* pair

$$d_i = (k_{d_i}, v_{d_i})$$

- k_{d_i} is a key that identifies the document (by abusive notation we noted i the key k_{d_i} in section 1 and 2;
- $v_{d_i} = \{a_{d_i,1} : v_{d_i,1}, \ldots, a_{d_i,n} : v_{d_i,n}\}$ is the document value. The document value v_{d_i} is defined as an *object* composed by a set of $(a_{d_i,j}, v_{d_i,j})$ pairs, where each $a_{d_i,j}$, is a *string* called *attribute* and each $v_{d_i,j}$, is the *value* that can be *atomic* (numeric, string, boolean, null) or *complex* (object, array). A value $v_{d_i,j}$ is defined below.

An atomic value is defined as follows $\forall j \in [1..n]$:

- $v_{d_i,j} = n$ if $n \in \mathbb{N}^*$, the set of numeric values *(integer, float)*;
- $v_{d_i,j} =$ "s" if "s" is a string formulated in $Unicode\mathbb{A}^*$;
- $v_{d_i,j} = b$ if $b \in B$, the set of boolean $\{true, false\}$;
- $v_{d_i,j} = \perp$ is a null value;

A complex value is defined as follows $\forall j \in [1..n]$:

- $v_{d_i,j} = \{a_{d_i,j,1} : v_{d_i,j,1}, \ldots, a_{d_i,j,m} : v_{d_i,j,m}\}$ is an object value where $v_{d_i,j,k}, \forall k \in [1..m]$ are *values*, and $a_{d_i,j,k}$, $\forall k \in [1..m]$ are Strings in $\mathbb{A}^*$ called *attributes*. This is a recursive definition identical to document value;
- $v_{d_i,j} = [v_{d_i,j,l}, \ldots, v_{d_i,j,l}]$ represents an array of values $v_{d_i,j,k}$, $\forall k \in [1..l]$, $l = \| v_{d_i,j} \|$;

In case of having document values $v_{d_i,j}$ as object or array, their inner values $v_{d_i,j,k}$ can be complex values too allowing to have different nesting levels. To cope with nested documents and navigate through schemas, we adopt the navigational path notations [3, 15].

Definition 4.3 (Schema). The schema, called s_{d_i}, is inferred from the document value $v_{d_i} = \{a_{d_i,1} : v_{d_i,1}, \ldots, a_{d_i,n} : v_{d_i,n}\}$ is defined as

$$s_{d_i} = \{p_1, \ldots, p_{m_i}\}$$

where p_j, $\forall j \in [1..m_i]$, is a path of each attribute of v_{d_i}, or navigational path for nested values such as $v_{d_i,j,k}$. For multiple nesting levels, the navigational path is extracted recursively to find the path from the root to the final atomic value that can be found in the document hierarchy.

A schema $s_{v_{d_i}}$ of value v_{d_i} from document d_i is formally defined as:

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

Example. Let us consider the documents d_1 and d_2 of Figure 2 . The underlying schema for both documents is described as follows:

$s_{v_{d_1}} = \{$"*movie_title*", "*year*", "*language*"$\}$
$s_{v_{d_2}} = \{$"*movie_title*", "*details*", "*details.year*", "*details.language*"$\}$

We notice that the attribute "*details*" from document d_2 is a complex one in which are nested the attributes "*year*" and "*language*" which leads to have two different navigational paths "*details.year*" and "*details.language*".

Definition 4.4 (Collection Schema). The schema S_C is inferred from collection C is defined by

$$S_C = \bigcup_{i=1}^{c} s_{v_{d_i}}$$

Definition 4.5 (Dictionary). The dictionary $dict_C$ of a collection C is defined by

$$\forall p_k \in S_C,\ dict_C = \{(p_k, \triangle_k)\}$$

- $p_k \in S_C$ is a path for an attribute which is present at least in one document of the collection;
- $\triangle_k = \{p_{p_{k,1}}, \ldots, p_{p_{k,q}}\} \subseteq S_C$ is a set of navigational paths leading to p_k;

For the rest of this paper, we will call equally any path p_k as $attribute$. We will use dictionary paths and dictionary attributes accordingly.

Example. The dictionary $dict_C$ constructed from the collection C is defined below, each dictionary entry p_k refers to the set of all extracted navigational paths.

$dict_C = \{$
$(movie\_title, \{movie\_title\})$,
(year, {year, details.year, versions.1.year, versions.2.year}),
$(language, \{language,\ details.language,$
$\quad versions.1.language, versions.2.language\})$,
$(details, \{details\})$,
$(details.year,\ \{details.year\})$,
$(details.language,\ \{details.language\})$,
$(versions,\ \{versions\})$,
$(versions.1,\ \{version.1\})$,
$(versions.1.year,\ \{versions.1.year\})$,
$(versions.1.language, \{versions.1.language\})$,
$(versions.2, \{versions.2\})$,
$(versions.2.year, \{versions.2.year\})$,
$(versions.2.language, \{versions.2.language\})$
$\}$
For example, the entry
$(year,\ \{year,\ details.year,\ versions.1.year,$
$\quad versions.2.year\})$ gives all navigational paths leading to the attribute "*year*".

4.2 Querying multi-structured data

Querying multi-structured data is possible via a combination of a set of unary operators. In this paper, we limit the querying process to projection and selection operators expressed by native MongoDB operators "$find$" and "$aggregate$".

4.2.1 Minimal closed kernel of unary operators. We define a minimal closed kernel of unary operators. We call C_{in} the queried collection, and C_{out} the resulting collection.

Definition 4.6 (Projection). The project operator helps to reduce initial schemas of documents from the collection to a finite sub-set of attributes as;

$$\pi_A(C_{in}) = C_{out}$$

where $A \subseteq S_{in}$ is a sub-set of attributes from $S_{C_{in}}$ (the schema of the input collection C_{in}

Definition 4.7 (Selection). The select operator runs to retrieve only documents that match some predicates; we call

$$\sigma_p(C_{in}) = C_{out}$$

where p refers to the predicate (or condition) for the selection operator. 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_i} is a value. It is possible to combine predicates by these operator from $\Omega = \{\vee, \wedge, \neg\}$ and this leads to a complex predicate.

We call $Norm_p$ the normal conjunctive form of the predicates p defined as follows:

$$Norm_p = \bigwedge_i \left(\bigvee_j a_{i,j}\ \varpi_{i,j}\ v_{i,j} \right)$$

We consider that all predicates in selection operators as in normal conjunctive form.

Definition 4.8 (Query). A query Q can be formulated by composing operators.

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

where $\forall i \in [1, r]\ q_i \in \{\pi, \sigma\}$

Example. Let us consider the collection presented in Figure 2.

$q_1 : \sigma_{(\text{“}language\text{”}=\text{“}English\text{”})}(C) = [$
{“movie_title” : “Fast and furious”, “year” : 2017,
“language” : “English”}
]
$q_2 : \pi_{(\text{“}movie\_title\text{”},\text{“}year\text{”})}(C) = [$
{“movie_title” : “Fast and furious”, year : 2017}
{“movie_title” : “Titanic”}
{“movie_title” : “Despicable Me 3”, year : 2017}
{“movie_title” : “The Hobbit”}
]
$q_3 : \pi_{(\text{“}movie\_title\text{”},\text{“}year\text{”})}(\sigma_{(\text{“}language\text{”}=\text{“}English\text{”})}(C)) = [$
{“movie_title” : “Fast and furious”, “year” : 2017}
]

Here, the query q_3 is constructed by combining *select* and *project* operators.

4.2.2 Query extension for multi-structured data. In this section, we introduce a new query extension algorithm that automatically enriches the user query. The native query engine of document-oriented stores such as MongoDB can efficiently execute our rewritten queries. Then, it is possible to find out all desired information regardless the structural heterogeneity inside the collection.

Algorithm 1: Automatic extension for 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 π_{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 σ_p : // selection
    do
      $P_{ext} \leftarrow \bigwedge_i \left(\bigvee_j \bigvee_{a_k \in \triangle_{i,j}}, a_k \varpi_{i,j} v_{i,j} \right)$
      $Q_{ext} \leftarrow Q_{ext} \circ \sigma_{P_{ext}}$
    end
  end
end

Our approach aims to enable transparent querying on a multi-structured collection of documents via automatic query rewriting. This process employs the materialized dictionary to enrich the original query by including the different navigational paths that lead to desired attributes. The algorithm 1 describes the query extension process as:

- In case of projection operation, the algorithm extends the list of attributes A_i by uniting different navigational paths $\triangle_k$ for each projected a_k.
- In case of the selection operation, the algorithm enriches the predicate p, expressed in the normal conjunctive form, with the set of extended dis-junctions built from the navigational paths $\triangle_{i,j}$ for each attribute $a_{i,j}$.

Example. Let us consider the query q_3 from the previous example. First, the query rewriting engine starts by extending the project operator. (line “*projection*” in Algorithm 1)

$$\pi_{(\text{“}movie\_title\text{”},\text{“}year\text{”})}(C)$$

For each projected field, the process consults the dictionary and extracts all the possible navigational paths. The dictionary entry, for the field *movie_title*, corresponds to:
($movie\_title$, {$movie\_title$})
So, A_{ext} = {$movie\_title$}
The dictionary entry, for the field *year*, corresponds to:
($year$, {$year$, $details.year$, $versions.1.year$, $versions.2.year$})
So, A_{ext} = {$movie\_title$, $year$, $details.year$, $versions.1.year$, $versions.2.year$}
The projection query is then rewritten as:

$$\pi_{(\text{“}movie\_title\text{”}, \text{“}year\text{”}, \text{“}details.year\text{”}, \text{“}versions.1.year\text{”}, \text{“}versions.2.year\text{”})}(C)$$

Next, the process continues with the selection query (line "selection" in Algorithm 1)

$$\sigma_{(\text{“}language\text{”}=\text{”}English\text{”})}(C)$$

The dictionary entry for the field *“language"* corresponds to:
($language$, {$language$, $details.language$, $versions.1.language$, $versions.2.language$})
So, P_{ext} = {
“language” = “English”
∨ “details.language” = “English”
∨ “versions.1.language” = “English”
∨ “versions.2.language” = “English”}

The selection is then rewritten as:

$$\sigma_{(\text{“}language\text{”}=\text{”}English\text{”}\vee\text{“}details.language\text{”}=\text{“}English\text{”}\vee\text{“}versions.1.language\text{”}=\text{“}English\text{”} \vee \text{“}versions.2.language\text{”}=\text{“}English\text{”})}(C)$$

Finally, the query rewriting engine generates the final query by combining the generated queries:

$$\pi_{(\text{“}movie\_title\text{”},\text{“}year\text{”},\text{“}details.year\text{”},\text{“}versions.1.year\text{”},\text{“}versions.2.year\text{”})}(\sigma_{(\text{“}language\text{”}=\text{“}English\text{”}\vee\text{“}details.language\text{”}=\text{“}English\text{”} \vee\text{“}versions.1.language\text{”}=\text{“}English\text{”}\vee\text{“}versions.2.language\text{”}=\text{“}English\text{”})}(C))$$
= [
{“movie_title” : “Fast and furious”, “year” : 2017},
{“movie_title” : “Titanic”, ”details” : {“year” : 2017}},
{“movie_title” : “The Hobbit”, “versions” : [
{“year” : 2017}]}
]

The query rewriting process injects additional complexity to the original user's queries.

5 EXPERIMENTS

In this section, we conduct a series of experiments to study the aforementioned points:

- Which are the effects on the execution time of the rewritten queries while varying the size of the collection and is this cost acceptable or not?
- Is the time to build the dictionary acceptable and what about the size of the dictionary according to structural variability?

Next, we explain the experimental protocol, then we study the queries execution cost, and finally we evaluate the dictionary generation time and its size.

5.1 Experimental protocol

We choose to run all the queries on synthetic datasets loaded into the document store MongoDB. In this section, we introduce the details of the experimental setup, the process of generating the synthetic datasets and the evaluation queries set. Later on, we present the results of executing the evaluation set in three separate contexts. The goal is to compare the cost of executing the rewritten queries; (i) the cost of executing the original queries on homogeneous documents, (ii) the execution time of several distinct queries that we build manually based on each schema. Then, we study the effects of the heterogeneity on the dictionary in terms of size and construction time. Finally, we evaluate the scale of the heterogeneity and its impact on generating the rewritten queries.

We conducted our experiments on MongoDB v3.4. We used an I5 3.4GHZ machine coupled with 16GB of RAM with 4 TB of storage space that runs CentOS7.

5.1.1 Dataset. To study the structural heterogeneity, we generate a custom synthetic datasets. First, we collected a JSON collection of documents from imdb[1] that describe movies. The original dataset has only flat documents with 28 attributes in each document. Then, we reuse this flat collection to produce documents with structural heterogeneity. For each generated dataset, we can 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 attributes. In other words, we cannot find the same grouping objects inside two structures. To make sure about the heterogeneity within documents, the grouping objects are unique in every schema. Only the original fields from the flat dataset are common to all documents. 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 structure. For the rest of the experiments, we built our dataset based on the characteristics that we describe in the Table 1. We generate collections of 10, 25, 50 and 100 GB of data.

Setting	Value
# of schema	10
# of grouping objects 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}
Percentage of schema presence	10%
# of attributes per schema	Random
# of attributes per grouping objects	Random

Table 1: Settings of the generated dataset

We generate a flat collection with same leaf attributes and their corresponding values as found in the heterogeneous datasets. This new collection helps us to have a proper environment and to compare; the execution time of the rewritten query on the heterogeneous datasets, versus the execution time of the original query on homogeneous datasets. Therefore, we ensure that every query returns the identical results from both heterogeneous or flat datasets. The same result implies: i) the same number of documents, and -0ii) the same values for their attributes (leaf fields).

5.1.2 Queries. we choose to build a synthetic set of queries based on the different comparison operators supported by MongoDB. We employed the classical comparison operators, i.e $\{<, >, \leq, \geq, =, \neq\}$ for numerical values as well as classical logical operators, i.e $\{and, or\}$ between query predicates. Also, we employed a regular expression to deal with string values. We select 8 attributes of different types and under different levels inside the documents in heterogeneous datasets. The Table 2 shows that for each attribute its type and the selection operator that we used later while formulating the synthetic queries. In addition, we present for each attribute the number of possible paths as found in the synthetic heterogeneous collection, the different nesting levels and the selectivity of the predicate.

Predicate	Attribute	Type	Operator	Paths	Depths	selectivity
p1	DirectorName	String	Regex{^A}	8	{8,2,3,9,6,5,4,7}	0,06 %
p2	Gross	Int	> 100 k	7	{7,8,2,3,9,6,4}	66 %
p3	Language	String	= "English"	7	{7,8,3,9,6,5,4}	0,018%
p4	Imdb_score	Float	<4,7	8	{8,7,2,3,4,5,6,9}	29 %
p5	Duration	Int	$\leq$ 200	7	{7,8,2,3,6,5,4}	77%
p6	Country	String	$\neq$ *Null*	6	{7,2,3,9,5,4}	100 %
p7	year	Int	< 1950	7	{7,8,2,3,6,5,4}	23 %
p8	FB_likes	Int	$\geq$ 500	7	{6,2,3,8,5,4,3}	83 %

Table 2: Query predicates

We formulate the following 6 queries:

- $Q1 : p1 \wedge p2$
- $Q2 : p1 \vee p2$

With the queries $Q1\&Q2$ the rewritten queries contain 15 predicates unlike the original queries that contains 2 predicates. 15 predicates are due to the 8 existing paths for $DirectorName$ in $p1$ plus 7 paths for $Gross$ in $p2$ that are included in a disjunctive form as described in rewriting algorithm.

- $Q3 : p1 \wedge p2 \wedge p5 \wedge p7$
- $Q4 : p1 \vee p2 \vee p5 \vee p7$

The rewritten versions of $Q3$ & $Q4$ contain 29 predicates unlike the original queries that contain 4 predicates.

- $Q5 : p1 \wedge p2 \wedge p5 \wedge p7 \wedge p6 \wedge p3 \wedge p4 \wedge p8$
- $Q6 : p1 \vee p2 \vee p5 \vee p7 \vee p6 \vee p3 \vee p4 \vee p8$

Finally, the rewritten versions of the queries $Q5\&Q6$ contain 57 predicates unlike the original queries that contain 8 predicates.

The Table 3 presents for each dataset: i) the number of documents inside the collection, ii) the number of expected results regarding each executed query.

Collection size in GB	# of documents	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6
10 GB	12 M	520 K	8,4 M	87,2 K	11,8 M	340	12 M
25 GB	30 M	1,3 M	21 M	218 K	29,5 M	850	30 M
50 GB	60 M	2.6 M	42 M	436 K	59 M	1,7 K	60 M
100 GB	120 M	5,2 M	84 M	872 K	118 M	3,4 K	120 M

Table 3: Number of extracted documents per query

[1] imdb.com

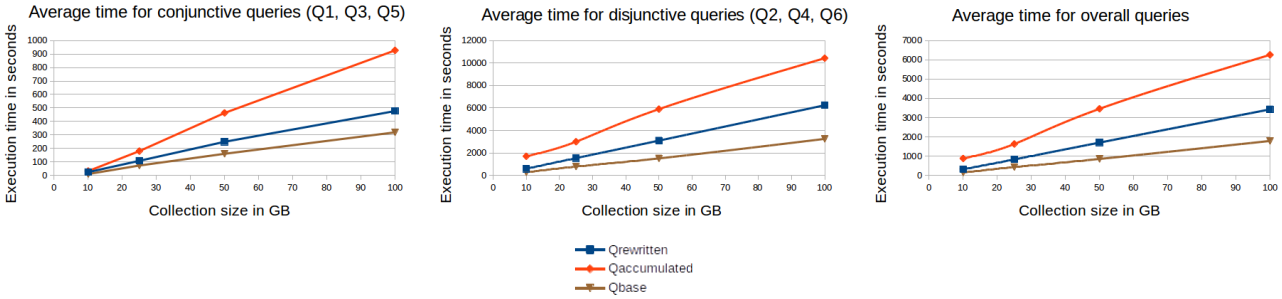

Figure 4: Query rewriting evaluations

Collection	Query	Usage
Homogeneous Flat documents	QBase user query	Baseline
Heterogeneous documents	QRewritten	Our solution
	QAccumulated the sum of the ten separated queries each one based on a specific schema.	"By hand" query context Used as a maximum cost line

Table 4: Evaluations context

5.2 Queries evaluation

5.2.1 Queries execution time. We define three contexts on which we run the above-defined queries. For each context, we measure the average of execution time after executing each query at least five times. The order of query execution is set to be random.

In the following, we present the details of the three evaluations contexts:

- We call "Q_{Base}" the query that refers to the initial user query (one of the above-defined queries), and it is executed over the homogeneous versions of the datasets. The purpose of this first context is to study the native behavior of the document store. We use this first context as a baseline for our experimentation.
- The "$Q_{Rewritten}$" refers to the query "Q_{Base}" rewritten by our approach. It is executed over the heterogeneous versions of the datasets.
- The "$Q_{Accumulated}$" refers to the set of equivalent queries formulated on each possible schema from the collection. In our case, it is mad of 10 separated queries since we are dealing with collections having ten schemas. These queries are build "by hand" as should have done any user without any assisting tool. We do not consider the time necessary to merge the results of each query as the goal is to compare the time to find the set of result documents. "$Q_{Accumulated}$" is obviously executed over the heterogeneous versions of the datasets.

Table 4 synthesizes our execution contexts.

As shown in Figure 4, we can notice that our rewritten query, $Q_{Rewritten}$, outperforms the accumulated one, $Q_{Accumulated}$. The difference between the two execution scenarios come from the capabilities of our rewritten query to include automatically all navigational paths extracted from the collection. Hence, the query is executed only once when the accumulated query may require several passes through the stored collection. This solution requires more CPU loads and more intensive disk I/O operations. We move now to study the efficiency of the rewritten query when compared to the baseline query Q_{Base}. We can notice that the overhead of our solution is up to two times (e.g., disjunctive form) when compared to the native execution of the baseline query on the homogeneous dataset. Moreover, we score an overall overhead that does not exceed 1,5 times in all six queries. We believe that this overhead is acceptable since we bypass the needed 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 include several navigational paths which will be processed by the native query engine of MongoDB to find matches in each visited document among the collection.

5.2.2 Dictionary and query rewriting engine at the scale. With this series of experiments, we try to push the dictionary and the query rewriting engine to their limits. To this end, we generated a heterogeneous synthetic collection of 1 GB. We use the primary 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 keep on the use of the query Q_6 introduced earlier in this section.

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

Table 5: Scale effects on query rewriting and dictionary size

We present the time needed to build the rewritten query in the Table 5. It is notable that the time to build the rewritten query is very low, less than two seconds. Also, it is possible to construct a dictionary over a heterogeneous collection of documents, here our dictionary can support up to 5 k of distinct schemas. The resulting size of the materialized dictionary is very encouraging since it does not require significant storage space. Furthermore, we also believe that the time spent to build the rewritten query is really interesting and represent another advantage of our solution. In this series of experiments, on each time we try to find distinct navigational paths for eight predicates. Each rewritten

query is composed by numerous disjunctive forms for each predicate. We notice 80 disjunctive forms while dealing with dataset having 10 schemas, 800 with 100 schemas, 8 k with 1 k schemas, 24 k with 3 k schemas and 40 k with 5k schemas. We believe that dictionary and the query rewriting engine scale well while dealing with heterogeneous collection of documents having an important number of schemas.

5.3 Dictionary construction time

In this part, we give the interest to the study of the dictionary constructions process. *EasyQ* offers the possibility to build the dictionary over existing dataset or during data loading phase. The dictionary contains the latest version of the data once all document are inserted. So, the query rewriting engine enrich the queries based on the new dictionary, otherwise if the process of data loading is in progress, it may do not take into account the recent changes. In the following, we study both configurations. First, we start by the evaluation of the time required to build the dictionary among pre-loaded five collections of 100 GB having 2, 4, 6, 8 and 10 schemas respectively.

We notice from the results in the Table 6 that the time elapsed to build the dictionary increases when we start to deal with collections having more heterogeneity. In case of the collection with 10 structures, the time does not exceed 40% when we compare it to a collection with 2 structures. We can again notice in the Table 6 the negligible size of the generated dictionaries when compared to the 100 GB of the collection.

# of schema	2	4	6	8	10
Required time (minutes)	96	108	127	143	156
Size of the resulting dictionary (KB)	4,154	9,458	13,587	17,478	22,997

Table 6: Time to build the dictionary of pre-loaded data

Afterwards, we give the interest to evaluate the overhead that causes the generation of the dictionary at loading time. We generate five collections of 1GB having the same structures from the last experiment (2, 4, 6, 8 and 10 schemas respectively). We present two measurements in Table 7. First, we measure the time to simply load each collection (without the dictionary building). Second, we measure the overall time to build the dictionary while loading the collection.

#of schemas	Load (s)	Load and dict. (s)	Overhead
2	201s	269s	33%
4	205s	277s	35%
6	207s	285s	37%
8	208s	300s	44%
10	210s	309s	47%

Table 7: Study of the overhead added during load time

In this experiments, we find that the overhead measure does not exceed 0.5 the time required to only load data. The evolution of the time while adding more heterogeneity is linear and not exponential which is encouraging. Many factors may affects the query construction phase. The number of attributes, the nesting levels may increase or decrease the overhead. The advantage our solutions is once the data is loaded and the dictionary is built or updated, the rewritten query takes automatically all changes into account.

6 CONCLUSION

NoSQL databases are often called schemaless because they may contain variable schemas among stored data. Nowadays, this variability is becoming a common standard in many applications as for example web applications, social media applications or internet of things world. Nevertheless, the existence of such structural heterogeneity makes it very hard for users to formulate queries to find out relevant and coherent results.

In this paper, to deal with structural heterogeneity, we suggest a novel approach for querying heterogeneous documents describing a given entity over NoSQL document stores. The developed tool is called *EasyQ*. Our objective is to allow users to perform their queries using a minimal knowledge about data schemas. *EasyQ* is based on two pillars. The first one is a dictionary that contains all possible paths for any existing 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. So, it is grounded on an important assumption: the collection describes homogeneous entities, i.e., a field has the same meaning in all document schemas. In case of ambiguity, the user should specify some sub-path in order to overcome the ambiguity. If this assumption is not guaranteed, users may face with irrelevant or incoherent results. Nevertheless this assumption may be acceptable in many applications, such as legacy collections, web applications or internet of things data.

In our first experiments, the evaluation consists in comparing the execution time cost of basic MongoDB queries and rewritten queries proposed by our approach. We conduct a set of tests by changing two primary parameters, the size of the dataset and the structural heterogeneity inside a collection (number of different schemas). Results show that the cost of executing rewritten queries proposed in this paper is higher when compared to the execution of basic user queries, but always less than twice. The overhead added to the performance of our query is due to the combination of multiple access paths to a queried field. Nevertheless, this time overhead is neglectful when compared to the execution of separated "*by hand*" queries for each schema while heterogeneity issues are automatically managed.

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 limitations regarding the number of paths and fields in the same query and regarding time cost. More experiments still to be performed on larger datasets and real case datasets. Another perspective is to enhance the current queries possibilities to introduce all existing classical operators of query languages (*contains*, *etc.*). It is also necessary to deal with other querying operators, particularly the aggregation operator.

The first long-term perspective consists in studying the real-time building of the dictionary when integrating data in order to take into account all possible queries: insert but also delete and update. It's likely the current simple structure of the dictionary will be transformed in depth to support more complex updates such as update and delete operations. The second long-term perspective consists in managing multi-store databases. The goal would be to extend the proposed approach to query data stored in different types of databases in a way independent from the various data schemas and stores. The final goal would be: how to query "*transparently*" any data store meanwhile being unaware of schemas or real fields names?

REFERENCES

[1] J Chris Anderson, Jan Lehnardt, and Noah Slater. 2010. *CouchDB: The Definitive Guide: Time to Relax.* " O'Reilly Media, Inc.".

[2] Mohamed-Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Schema inference for massive json datasets. In *Extending Database Technology (EDBT)*.

[3] Pierre Bourhis, Juan L Reutter, Fernando Suárez, and Domagoj Vrgoč. 2017. JSON: data model, query languages and schema specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 123–135.

[4] Max Chevalier, Mohammed El Malki, Arlind Kopliku, Olivier Teste, and Ronan Tournier. 2015. Implementation of multidimensional databases in column-oriented NoSQL systems. In *East European Conference on Advances in Databases and Information Systems*. Springer, 79–91.

[5] Kristina Chodorow and Michael Dirolf. 2010. MongoDB: The Definitive Guide O'Reilly Media. (2010).

[6] Mohamed Lamine Chouder, Stefano Rizzi, and Rachid Chalal. 2017. Enabling Self-Service BI on Document Stores.. In *EDBT/ICDT Workshops*.

[7] Alejandro Corbellini, Cristian Mateos, Alejandro Zunino, Daniela Godoy, and Silvia Schiaffino. 2017. Persisting big-data: The NoSQL landscape. *Information Systems* 63 (2017), 1–23.

[8] Avrilia Floratou, Nikhil Teletia, David J DeWitt, Jignesh M Patel, and Donghui Zhang. 2012. Can the elephants handle the nosql onslaught? *Proceedings of the VLDB Endowment* 5, 12 (2012), 1712–1723.

[9] Daniela Florescu and Ghislain Fourny. 2013. JSONiq: The history of a query language. *IEEE internet computing* 17, 5 (2013), 86–90.

[10] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. 2003. XRANK: Ranked keyword search over XML documents. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 16–27.

[11] Rihan Hai, Sandra Geisler, and Christoph Quix. 2016. Constance: An intelligent data lake system. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2097–2100.

[12] Michael Hausenblas and Jacques Nadeau. 2013. Apache drill: interactive ad-hoc analysis at scale. *Big Data* 1, 2 (2013), 100–104.

[13] Robin Hecht and Stefan Jablonski. 2011. NoSQL evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*. IEEE, 336–341.

[14] Victor Herrero, Alberto Abelló, and Oscar Romero. 2016. NOSQL design for analytical workloads: variability matters. In *Conceptual Modeling: 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings 35*. Springer, 50–64.

[15] Jan Hidders, Jan Paredaens, and Jan Van den Bussche. 2017. J-Logic: Logical Foundations for JSON Querying. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 137–149.

[16] Prashant R Lambole and Prashant N Chatur. 2017. A review on XML keyword query processing. In *Innovative Mechanisms for Industry Applications (ICIMIA), 2017 International Conference on*. IEEE, 238–241.

[17] Yunyao Li, Cong Yu, and HV Jagadish. 2004. Schema-free xquery. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 72–83.

[18] Liang Lin, Vera Lychagina, Weiran Liu, Younghee Kwon, Sagar Mittal, and Michael Wong. 2011. Tenzing a sql implementation on the mapreduce framework. (2011).

[19] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2014. The SQL++ query language: Configurable, unifying and semi-structured. *arXiv preprint arXiv:1405.3631* (2014).

[20] Pavel Shvaiko and Jérôme Euzenat. 2005. A survey of schema-based matching approaches. *Journal on data semantics IV* (2005), 146–171.

[21] M. Stonebraker. 2012. New opportunities for New SQL. *Commun. ACM* 5, 11 (2012), 10–11.

[22] Daniel Tahara, Thaddeus Diamond, and Daniel J Abadi. 2014. Sinew: a SQL system for multi-structured data. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 815–826.

[23] Lanjun Wang, Shuo Zhang, Juwei Shi, Limei Jiao, Oktie Hassanzadeh, Jia Zou, and Chen Wangz. 2015. Schema management for document stores. *Proceedings of the VLDB Endowment* 8, 9 (2015), 922–933.

[24] Rui Zhou, Chengfei Liu, and Jianxin Li. 2010. Fast ELCA computation for keyword queries on XML data. In *Proceedings of the 13th International Conference on Extending Database Technology*. ACM, 549–560.