Schema-independent querying for heterogeneous collections in NoSQL document stores

Hamdi Ben Hamadou [a,*], Faiza Ghozzi [b], André Péninou [a], Olivier Teste [a]

[a] *IRIT, Université de Toulouse, CNRS, 118 route de Narbonne, 31400 Toulouse, France*
[b] *MIRACL, Université de Sfax, ISIMS, Route de Tunis Km 10 B.P. 242, 3021 Sfax, Tunisie*

HIGHLIGHTS

- Document stores offer the flexibility to store documents with heterogeneous schemas.
- Querying document stores requires complex queries to overcome schemas heterogeneity.
- We propose to build queries over partial paths, regardless of documents schemas.
- We formally define the reformulation of such queries for most document operators.
- Queries are extended using a dictionary to bind paths to all existing absolute paths.

ARTICLE INFO

ABSTRACT

NoSQL document stores are well-tailored to efficiently load and manage massive collections of heterogeneous documents without any prior structural validation. However, this flexibility becomes a serious challenge when querying heterogeneous documents, and hence the user has to build complex queries or reformulate existing queries whenever new schemas are introduced in a collection. In this paper we propose a novel approach, based on formal foundations, for building schema-independent queries which are designed to query multi-structured documents. We present a query enrichment mechanism that consults a pre-constructed dictionary. This dictionary binds each possible path in the documents to all its corresponding absolute paths in all the documents. We automate the process of query reformulation via a set of rules that reformulate most document store operators, such as select, project, unnest, aggregate and lookup. We then produce queries across multi-structured documents which are compatible with the native query engine of the underlying document store. To evaluate our approach, we conducted experiments on synthetic datasets. Our results show that the induced overhead can be acceptable when compared to the efforts needed to restructure the data or the time required to execute several queries corresponding to the different schemas inside the collection.

1. Introduction

During the last decade, NoSQL databases and schemaless data modelling have emerged as mainstream alternatives to relational modelling for addressing the substantive requirements of current data-intensive applications [1], e.g., IoT, web, social media and logs. Document stores hold data in collections of documents (most often JSON objects); they do not require the definition of any formal structure before loading data, and any data structure can be used when updating data. The main advantage of this is being able to store documents in transparent and efficient

ways [2,3]. Nevertheless, it is possible to store a set of heterogeneous documents inside the same collection, and for the purposes of this paper, documents have heterogeneous structures. This is a major drawback, and issues arise when querying such data because the underlying heterogeneity has to somehow be resolved in the query formulation in order to provide relevant results. Several kinds of heterogeneity are discussed in the literature: structural heterogeneity refers to diverse representations of documents, e.g., nested or flat structures, nesting levels, etc. as shown in Fig. 1; syntactic heterogeneity refers to differences in the representation of data, and specifically of attribute names, e.g., *movie_title* or *movieTitle*; finally, semantic heterogeneity may exist when the same field relies on distinct concepts in separate documents [4], e.g., *country* and *nation*. In this paper we focus on structural heterogeneity.

Usually, four main types of solution are considered when dealing with structural heterogeneity: (i) applying schema matching

* Corresponding author.
E-mail addresses: hamdi.ben-hamadou@irit.fr (H. Ben Hamadou),
faiza.ghozzi@isims.usf.tn (F. Ghozzi), andre.peninou@irit.fr (A. Péninou),
olivier.teste@irit.fr (O. Teste).

techniques to merge heterogeneous structures [5], (ii) transforming all document schemas to a single common schema which leads to a homogeneous collection [6], (iii) using automatic schema discovery from original heterogeneous data to support querying, which requires the user to take heterogeneity into account [7], and (iv) introducing a new querying mechanism to give transparency to the heterogeneity in the data [8].

In this paper we provide an automatic mechanism based on formal foundations so that multi-structured document stores can be queried. We support the user query a collection based on the knowledge of absolute, i.e., a full path leading to the attribute of interest starting from the root of the document, or partial paths, i.e., a sub-path starting from any intermediary location within the document leading to the attribute of interest, that exist in some schema. Our query reformulation engine generates a new query that automatically integrates the description of heterogeneous structures. Therefore, the engine enriches path expression in the query with all the absolute paths that exist in any document from the collection and that lead to this path expression.

The rest of our paper is structured as follows. In Section 2 we examine the main issues addressed in this paper. Section 3 reviews the state-of-the-art research work which provides support for querying multi-structured documents. Section 4 describes our approach in detail. Section 5 presents our first experiments and evaluates our approach. Finally, we summarize our findings in Section 6.

2. Issues in querying multi-structured documents

In this section we outline the issues that arise when querying multi-structured documents.

2.1. Structural heterogeneity

Classically, a collection contains a set of documents that usually represent the same entity. Nevertheless, because of their flexibility, document stores can store documents inside the same collection regardless of their structure. Such schema variability appears as applications evolve and change for many reasons: systems evolution, systems maintenance, diversity of data sources, data enrichment over time, etc. Because of their schemaless nature, querying document stores requires a knowledge of the underlying document structure and the use of full paths to access data in queries. Most document stores adopt this assumption (e.g., MongoDB, CouchDB, Terrastore [9–11]). Since it is possible to construct a collection where documents are structurally heterogeneous, users can therefore formulate queries which affect the quality of the results, i.e., by excluding relevant documents whose schema is not explicitly expressed in the query.

In Fig. 1 we present a collection composed of four documents (a, b, c, d) using the JSON (JavaScript Object Notation) format for formatting semi-structured data in human-readable text. Each document contains a set of attribute-value pairs whose values can be simple (atomic), e.g., the value of the attribute *title*, or complex, e.g., the value of the attribute *ranking* in document (a). A special attribute *_id* in each document identifies the document inside the collection. In addition, documents form a hierarchical data structure composed of several nesting levels (also called nodes or attributes), e.g., the attribute *score* in document (a) is nested under the complex attribute *ranking*. The top node for all attributes in the document is called the root, but has no specific name. Fig. 2 illustrates the hierarchical representation of document (a).

In collection (C), the documents (b, c, d) share the same leaf nodes (attributes with atomic/array of atomic values, e.g., *title*, *genres*) as document (a). The structural heterogeneity lies in the fact that these leaf nodes exist in different locations in documents (b, c, d), for instance, the absolute path to reach the attribute *title* in document (c) is *film.title*. However, in documents (a, b), the path *title* is enough to reach this information because it is directly nested under the root node. Nevertheless, *film.title* represents a third absolute path in document (c) and *description.title* represents a fourth absolute path in document (d) for the *title* information.

2.2. Querying issues

To retrieve information from a document attribute in most existing document stores, it is necessary to build queries using the absolute path from the document root down to the attribute of interest. If a user formulates a projection query using only the absolute path *title*, any document store engine ignores the information related to this attribute in documents (c) and (d), despite the fact it is present in those documents. As a result, document stores return only *{"_id":1, "title":"Million Dollar Baby"}, {"_id":2, "title":"In the Line of Fire"}*. This result is closely related to the paths expressed in the query. Because the majority of NoSQL document stores require the use of absolute paths, when a user makes a query, native query engines expect this user to explicitly include all existing paths from the database to target the relevant data.

It is not a straightforward task to handle structural heterogeneity manually, especially in continuously evolving big data contexts where data variety is quite common, for instance, to project out all information related to the attribute *year*, the user should know about the distinct absolute paths found in collection (C) (i.e., *year, info.year, film.details.year, description.year*) otherwise the resulting information could be reduced.

Let us suppose that a user wishes to project out all information related to movies: *title* with their related *ranking.score*. If she formulates a query with the paths (*title, ranking.score*) the result is *{"_id":1, "title":"Million Dollar Baby", {"ranking":{"score":8.1}}, {"_id":2, "title":"In the Line of Fire"}*.

Despite the presence of the information *ranking.score* in the four documents, the result does not include this information since it is located in other paths in documents (b, c, d). We can also see the same behaviour for the attribute *title* with documents (c, d). Let us assume that the user knows the absolute path for *ranking.score* in document (b) and formulates a second query with the paths (*title, info.ranking.score*), in this case the result is *{"_id":1, "title":"Million Dollar Baby"}, {"_id":2, "title":"In the Line of Fire", {"info":{"ranking":{"score":8.1}}}*.

When we compare the results of the two previous queries, we can observe that information related to *ranking.score* for document (a) is only present for the first result. However the second query just retrieves *ranking.score* information from document (b). Formulating and executing several queries is a complex and an error prone-task. Data redundancy may occur (case of *title* information present in both results). Therefore, if a user wishes to query multi-structured data and use several queries to target different paths, she has to make an effort to merge results, to learn the underlying data structures, and to remove possibly redundant information. In this example, a possible query allowing the user to consider all the data could take the following form (*title, film.title, description.title, ranking.score, info.ranking.score, film.others.ranking.score, classification.ranking.score*) which is a long and complex query for projecting out only two pieces of information, i.e., *title* and *ranking.score*.

```
{  "_id":1,
   "title":"Million Dollar Baby",
   "year":2004,
   "link":null,
   "awards":["Oscar",  "Golden Globe",
      "Movies for Grownups Award", "AFI
      Award"],
   "genres":["Drama", "Sport"],
   "country":"USA",
   "director":{ "first_name":"Clint",
      "last_name":"Eastwood"
   },
   "lead_actor":{ "first_name":"Clint",
      "last_name":"Eastwood"
   },
   "actors":["Clint Eastwood",
      "Hilary Swank", "Morgan Freeman"],
   "ranking":{ "score":8.1
   }
}
```

(a)

```
{  "_id":2,
   "title":"In the Line of Fire",
   "info":{
      "year":1993,
      "country":"USA",
      "link":"https://goo.gl/2A253A",
      "genres":["Drama", "Action", "Crime"],
      "people":{
         "director":{ "first_name":"Clint",
            "last_name":"Eastwood"
         },
         "lead_actor":{ "first_name":"Clint",
            "last_name":"Eastwood"
         },
         "actors":["Clint Eastwood",
            "John Malkovich", "Rene Russo Swank"]
      },
      "ranking":{ "score":7.2
      }
   }
}
```

(b)

```
{  "_id":3,
   "film":{
      "title":"Gran Torino",
      "awards": "AFI Award",
      "link":null,
      "details":{
         "year":2008,
         "country":"USA",
         "genres":"Drama",
         "director":{ "first_name":"Clint",
            "last_name":"Eastwood"
         },
         "personas":{
            "lead_actor":{ "first_name":"Clint",
               "last_name":"Eastwood"
            },
            "actors":["Clint Eastwood",
               "Bee Vang", "Christopher Carley"]
         }
      },
      "others":{
         "ranking":{ "score":8.1
         }
      }
   }
}
```

(c)

```
{  "_id":4,
   "description":{
      "title":"The Good, the Bad and the Ugly",
      "year":1966,
      "link":"goo.gl/qEFfUB",
      "country":"Italy",
      "director":{ "first_name":"Sergio",
         "last_name":"Leone"
      },
      "stars":{
         "lead_actor":{ "first_name":"Clint",
            "last_name":"Eastwood"
         },
         "actors":["Clint Eastwood",
            "Eli Wallach", "Lee Van Cleef"]
      }
   },
   "classification":{
      "ranking":{"score":7.2
      },
      "genres":["Western"]
   }
}
```

(d)

Fig. 1. Illustrative example of a collection (C) with four documents describing films.

Table 1
Comparative study of the main contributions to querying heterogeneous semi-structured data.

| Contribution	Heterogeneity		Querying mechanism	Underlying store	Solution	Data model	Schema evolution support
	Type	Level					
ARGO [12]	Structural	Schema	ARGO/SQL	MySQL, Postgres	Physical re-factorization	Document	Manual
Sinew [6]	Structural	Schema	SQL	Postgres	Physical re-factorization	Key–value	Manual
[13]	semantic	Schema	SQL	RDBMS	Physical re-factorization	Key–value	Manual
[14]	Structural	instance	Keywords queries + SQL	–	Data annotation	Document	Manual
[15]	Structural	Schema	–	Distributed DB	Schema inference	Document	Manual
[16]	Structural	Schema	–	MongoDB	Schema inference	Document	Manual
SQL++[17]	Structural	Schema	SQL++	RDBMS+NoSQL	Query language	Relational + document	Manual
JSONiq [8]	Structural	Schema	JSONiq	–	Query language	Document	Manual
XQuery [18]	Structural	Schema	XQuery	–	Query language	Document	Manual
EasyQ	Structural	Schema	Aggregation Framework	MongoDB	Query reformulation	Document	Automatic
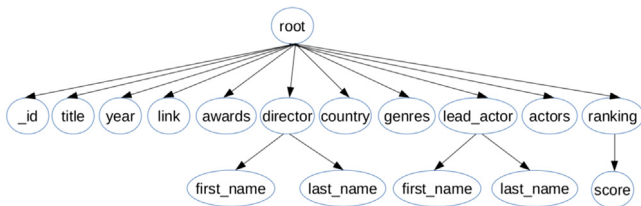
Fig. 2. Hierarchical representation of the document (a).

3. Related work

Contexts such as data-lake [14], federated database [19], data integration, schema matching [5], and recently, schemaless data support in NoSQL systems [20] have highlighted the importance of building transparent mechanisms that use the underlying data in a transparent way. In addition to large volumes of data, there is a need to overcome the heterogeneity of the collected data. Different sources generate data under different structures, versions and languages. The problem of querying multi-structured data has pushed the database community to rethink how information is accessed with regards to the underlying data structure heterogeneity [21].

We classify state-of-the-art research work based on the solutions proposed for querying multi-structured documents. The first family of work examines methods of schema matching; the second recommends performing materialized structural changes to unify heterogeneous forms of documents; the third recommends operating queries on a virtual schema derived from the heterogeneous structures and the last recommends querying techniques to overcome the heterogeneity in documents.

Schema integration. The schema integration process is performed as an intermediary step to facilitate a query execution process. In their survey paper [5], the authors presented the state-of-the-art techniques used to automate the schema integration process. Matching techniques can cover schemas [22] or even instances [23]. Traditionally, lexical matches are used to handle syntactic heterogeneity [24]. Furthermore, thesauruses and dictionaries are used to perform semantic matching [25]. Schema integration techniques may present certain issues such as data duplication, e.g., in the case of physical re-factorization, or potential original structure loss, e.g., when constituting a common schema. These two characteristics may make it impossible or unacceptable to support legacy applications. Therefore, changing the data structure necessitates changing the workload in the application side. Furthermore, this task is required whenever a new common structure is integrated into the collection data. It is important to note that our schema-independent querying manages to resolve heterogeneity at the schema level using schema level matching techniques. We do not consider instances and values in our approach, only the document structures.

Physical re-factorization. Work has been conducted to ensure that semi-structured data can be queried without any prior schema validation or restriction. Generally, this work recommends flattening XML or JSON data into a relational form [12,6,13]. SQL queries are formulated based on relational views built on top of the inferred data structures. This strategy implies that several physical re-factorizations should be performed which will affect scalability. Hence, this process is time-consuming, and it requires additional resources, such as an external relational database and more effort from the user to learn the unified inferred relational schema. Users of these systems have to learn new schemas every time they change the workload or when new data are inserted (or updated) in the collection, as this is necessary to regenerate the relational views and stored columns after every change.

Schema inference. Other research work recommends inferring implicit schemas from semi-structured documents. The idea is to give an overview of the different elements present in the integrated data [15,16]. In [7] the authors suggest that all document schemas should be summarized in a skeleton framework to discover the existence of fields or sub-schemas inside the collection. In [26] the authors suggest extracting collection structures to help developers in the process of designing their applications. In [27] a novel technique is defined to explain the schema variants within a collection in document stores. Therefore, the heterogeneity problem in this research work is detected when the same attribute is represented differently (different type, different position inside documents). Schema inference methods enable the user to have an overview of the data and take the necessary measures and decisions during the application design phase. The limitation with such a logical view is that it requires a manual process in order to build the desired queries by including the desired attributes and all their possible navigational paths. In such approaches, the user is aware of data structures but is required to manage the heterogeneity.

Querying techniques. Other work recommends resolving the heterogeneity problem by focusing on the query side. Query rewriting [28] is a strategy for reformulating an input query into several derivations to overcome heterogeneity. The majority of research work is designed in the context of the relational database, where heterogeneity is usually restricted to the lexical level. When it comes to the hierarchical nature of semi-structured data (XML, JSON documents), the problem of identifying similar nodes is insufficient to resolve the problem of querying documents with structural heterogeneity. To this end, keyword querying has been adopted in the context of XML [29]. The process of answering a keyword query on XML data starts with the identification of 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 heterogeneity in terms of nodes, but assume that if the keyword is found, no matter what its containing node is, the document is adequate and has to be returned to the user. Other alternatives for finding different navigational paths which lead to the same nodes are supported by [30,18]. However, structural heterogeneity is only partially addressed. There is always a need to know the underlying document structures and to learn a complex query language. Moreover, these solutions are not built to run with large-scale data. In addition, we can see the same limitations with JSONiq [8], the extension to XQuery designed to deal with large-scale semi-structured data.

In Table 1 we present the state-of-the-art research work intended to resolve the problem of querying multi-structured data. We compare this work according to the following criteria:

- The type of heterogeneity examined in each type of work: structural, syntactic or semantic;
- The level of heterogeneity. For each type of work we consider whether the contribution is designed to resolve heterogeneity at schema level or instance level;
- The querying mechanism. We examine if the type of work recommends a new query language, reuses existing systems or does not offer any querying support;
- The underlying store. We indicate if each type of work is limited to one store or several stores;
- The solution proposed for the heterogeneity problem. We describe the nature of the solution for each type of work, for instance, does it perform physical re-factorization and change the original schemas, does it focus only on inferring underlying schemas or does it offer a new query language?

- The data models. We classify work according to the data models it supports: documents, key–value, relational, etc.;
- Schema evolution support. We indicate how each type of work handles the arrival of new data structures in the database (insert/update/delete documents). Do they offer transparent and native support to handle these new structures? Are manual changes needed to support this change?

Similarly to our work, the majority of the state-of-the-art research concentrates on managing heterogeneity at a structural level. If we take into account schema evolution support, to the best of our knowledge our work is the first contribution that manages automatic support to overcome structural heterogeneity without regenerating relational views or re-executing schema inference techniques. Moreover, our contribution is able to automatically extract existing schemas, build and update a dictionary with all the details of the attributes and their corresponding paths in the collection, and offer querying capabilities without introducing a new querying language or new store. In addition, we built our query reformulation mechanism based on ideas introduced in previous work designed for this purpose. We propose to help the user overcome heterogeneity: she queries the system with a minimum knowledge of the data structures and the system reformulates the query to overcome the underlying heterogeneity. We ensure that our query reformulation is able to reformulate queries with the latest schemas in the collection.

This paper introduces a schema-independent querying approach that is based on the native engine and operators supported by conventional document stores. Furthermore, we offer greater support for most querying operators, e.g., project-select-aggregate-unnest-lookup to enhance the basic querying capabilities of our previous work which only supports project, select and aggregate operators [21,31]. Our approach is an automatic process running on the initial document structures; there is no need to perform any transformation to the underlying structures or to use further auxiliary systems. Users are not asked to manually resolve the heterogeneity. For collections of heterogeneous documents describing a given entity, we believe that we can handle the structural heterogeneity of documents by using a query reformulation mechanism introduced in this paper.

4. Easy schema-independent querying for heterogeneous collections in NoSQL document stores

In this section we give an in-depth description of the key component of our approach, *EasyQ* (*Easy Query*), with reference to a series of formal backgrounds, in particular, the data model and the querying operators.

Most document-oriented databases do not offer native mechanisms which enable schema-independent querying. Schemaless flexibility is ensured during the loading stages. However, querying multi-structured collections becomes challenging.

Let us consider the collection (C) from Fig. 1 and let us suppose that a user wishes to project the attribute *year*. There are four distinct locations for the attribute *year* in documents: root-based for document (a), but as a leaf-node for documents (b, c, d). Hence, the simple and classical query (*year*) can only reach *year* information in document (a) since document stores only accept absolute paths (root-based) in queries.

To overcome the structural heterogeneity we propose an automatic mechanism that reformulates the initial query as a new query where all corresponding locations for the attribute *year* are considered. The attribute *year* is reachable using the following four distinct absolute paths: (*year, info.year, film.details.year, description.year*). Our mechanism produces a new projection operation that contains all these absolute paths.

To facilitate the task of path discovery we introduce a dictionary that contains the set of corresponding absolute paths in all document structures for any partial path (sub-path included in an existing absolute path), any leaf node and any absolute path in the collection. During the data loading stage, the system parses the underlying document structures and creates new dictionary entries or updates existing ones with their corresponding absolute paths. For instance, in the case of collection (C), to reformulate the projection operation, our query reformulation engine finds the following entry in the dictionary (*year,[year, info.year, film.details.year, description.year]*) and reformulates the query.

As previously highlighted most document stores only accept queries which have been formulated for absolute paths. In our proposal we offer the possibility of querying a collection using *partial paths* that are not necessarily root-based and could not end with leaf nodes. Furthermore, the use of partial paths helps the user to disambiguate her query when ambiguous entities appear in documents. For example, a single leaf node attribute may refer to various semantics when used for different objects in the same document, and thus relates to different entities. Let us consider the attribute *first_name* in document (a). It is found under *director,* and in this case the semantic attribute is related to the director of the film and under *lead_actor* the semantic attribute is related to the main actor in the film. When the user tries to execute the following projection operation (*director.first_name*), the native query engine considers an absolute path and will only return document (a). With *EasyQ*, when we search for *director.first_name* in the dictionary we will find the following entry: (*director.first_name: [director.first_name, info.people.director.first_name, film.details.director.first_name, description.director.first_name]*). The reformulation hence leads to the projection (*director.first_name*, *info.people.director.first_name*, *film.details.director.first_name*, *description.director.first_name*) and the result contains information related to *director.first_name* for all documents (a, b, c, d). The query reformulation therefore overcomes heterogeneity, and the use of partial paths considerably helps the user resolve the ambiguous entity problems when formulating their queries.

The presence of all existing partial paths and leaf node as keys in the dictionary enables users to freely express their query in order to use the different semantics embedded in document structures. When a user formulates a query with leaf nodes, EasyQ returns all the information for each leaf node, regardless of their semantics. However, when the user indicates partial or absolute paths, EasyQ only returns information related to the explicit semantic expressed by these paths.

In the next sections we give an in-depth description of the key components of our approach, *EasyQ* (*Easy Query*), with reference to a series of formal backgrounds, in particular, the data model and the querying operators.

4.1. Architecture overview

Fig. 3 provides a high-level illustration of the architecture of *EasyQ* with its two main components: the query reformulation engine and the dictionary. Moreover, Fig. 3 shows the flow of data during the data loading stage and the query processing stage.

We introduce the data structure extractor during the data loading phase. It enriches the dictionary with new partial path entries and updates existing ones with corresponding absolute paths in documents. 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, denoted by *Q*, which is formulated using any combination of paths
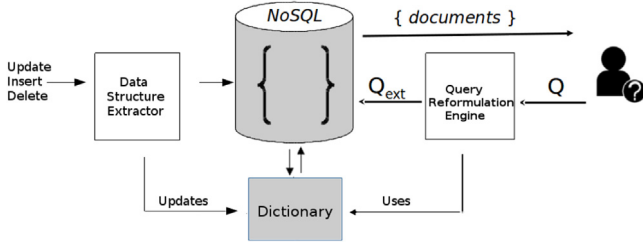
Fig. 3. EasyQ architecture: data structure extractor and query reformulation engine.

(leaf nodes, partial paths and absolute paths) and the desired collection. The *EasyQ* query reformulation engine reads from the dictionary and produces an enriched query known as Q_{ext}, that includes all existing absolute paths from all the documents. Finally, the document store executes Q_{ext} and returns the result to the user.

In the remainder of this section we describe the formal model of multi-structured documents, the dictionary and the queries across multi-structured documents, and the reformulation rules for each operator.

4.2. Formal foundations

In this section we introduce the formal foundations that cover the basic definition of the document and collection concepts.

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

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

where $n_c = |C|$ is the collection size.

Definition (Document). A document $d_i \in C$, $\forall i \in [1, n_c]$, 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,
- v_{d_i} is the *document value*.

We first start by defining a generic value v which can be *atomic* or *complex* (object or array). An atomic value v can take one of the following four forms:

- $v = n$ where n is a numerical value form (*integer* or *float*);
- $v = $ "s" where "s" is a string formulated in *Unicode* $\mathbb{A}^*$;
- $v = \beta$ where $\beta \in B$, the set of boolean $B = \{True, False\}$;
- $v = \perp$ where $\perp$ is the *null* value.

A complex value v can take one of the following two forms:

- $v = \{a_1 : v_1, \ldots, a_m : v_m\}$ is an object value, $m = |v|$ and $\forall j \in [1..m]$, v_j are *values*, and a_j are strings (in *Unicode* $\mathbb{A}^*$) called *attributes*. This definition is recursive since a value v_j is defined as a generic value v;
- $v = [v_1, \ldots, v_m]$ represents an array of $m = |v|$ values v_j and $\forall j \in [1..m]$ v_j are *values*. This definition is also recursive because a value v_j is defined as a generic value v.

We use the definition of generic value v to define: (i) the document value v_{d_i} composed of a set of attribute $a_{d_i,j}$ value $v_{d_i,j}$ pairs, (ii) the attribute value $v_{d_i,j}$, $j \in [1..m]$. Therefore, in the event of complex attribute values $v_{d_i,j}$, i.e., object or array, their internal values $v_{d_i,j,k}$, $k \in [1..|v_{d_i,j}|]$, can also be complex and they can take the same form as *generic value* v (atomic or complex). To cope with nested documents and navigate through schemas, we adopt classical navigational path notations [32,33].

Definition (Path). A path represents a sequence of dot concatenated attributes starting from the root of the document and leading to a particular attribute in the document value v_{d_i} that could be an atomic value of a leaf node or a complex value of a document object. In both cases, the path from the root to any atomic or complex document value in v_{d_i} is called an *absolute path*, e.g., the path *ranking.score* in document (a) represents an absolute path to reach the information referenced by the attribute *score*. Likewise, the path *info.people.actors* in document (b) is an absolute path. Furthermore, a path could be a *sub-path* when the sequence of attributes does not start from the root. In this case, the path is called a *partial path*, e.g., the path *lead_actor.first_name* in documents (b, c, d) represents partial paths which reach the information referenced by the attribute *first_name* of the *lead_actor*. Likewise, *people.director* in document (b) is a partial path. Finally, leaf node attributes are considered as paths too since they respond to the *partial* path definition.

Definition (Document Schema). The document schema S_{d_i} inferred from the document value v_{d_i} from document d_i, is defined as:

$$S_{d_i} = \{p_1, \ldots, p_{N_i}\}$$

where, $N_i = |S_{d_i}|$ and $\forall j \in [1..N_i]$, p_j is an absolute path leading to an attribute of v_{d_i}. For multiple nesting levels, the navigational paths are extracted recursively in order to find the path from the root to any attribute that can be found in the document hierarchy. The schema S_{d_i} of a document d_i is defined from its value $v_{d_i} = \{a_{d_i,1} : v_{d_i,1}, \ldots, a_{d_i,n_i} : v_{d_i,n_i}\}$ as follows:

- if $v_{d_i,j}$ is atomic, $S_{d_i} = S_{d_i} \cup \{a_{d_i,j}\}$ where $a_{d_i,j}$ is a path leading to the value $v_{d_i,j}$;
- if $v_{d_i,j}$ is an 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 $v_{d_i,j}$ and $a_{d_i,j}.p$ is a path composed of the complex attribute $a_{d_i,j}$ dot concatenated with the path p extracted from $s_{d_i,j}$ leading to the internal values of $v_{d_i,j}$;
- if $v_{d_i,j}$ is an array, $S_{d_i} = S_{d_i} \cup \{a_{d_i,j}\} \cup \{ \cup_{k=1}^{m_j} \left(\{a_{d_i,j}.k\} \cup \{\cup_{p \in s_{d_i,j,k}} a_{d_i,j}.k.p\} \right)\}$ where $s_{d_i,j,k}$ is the schema of the kth value in the array $v_{d_i,j}$, $a_{d_i,j}.k.p$ is a path leading to the kth entry from the array value $v_{d_i,j}$ composed of the array attribute $a_{d_i,j}$ dot concatenated with the index k and dot concatenated with the path p extracted from $s_{d_i,j,k}$; we adopt this notation from [33].

Example. The document schema for the document (b) is as follows:

$$S_b =$$

```
{_id,                      info.people.director.first_name,
title,                     info.people.director.last_name,
info,                      info.people.lead_actor,
info.year,                 info.people.lead_actor.first_name,
info.country,              info.people.lead_actor.last_name,
info.link,                 info.actors,
info.genres,               info.actors.1,
info.genres.1,             info.actors.2,
info.genres.2,             info.actors.3,
info.genres.3,             info.ranking,
info.people,               info.ranking.score}
info.people.director,
```

Definition (Collection Schema). The schema S_C inferred from a collection C is the set of all absolute paths defined in each document schema extracted from each document in the collection C:

$$S_C = \bigcup_{i=1}^{n_c} S_{d_i}$$

4.3. Dictionary

The architecture of our approach relies on the construction of a dictionary that enables the query reformulation process. A dictionary is a repository that binds each existing path in the collection (partial or absolute paths, including leaf nodes) to all the absolute paths from the collection schema leading to it.

In the following paragraphs we first define partial paths in documents (called document paths), then partial paths in the collection (called collection paths) and we finally give the formal definition of the dictionary.

Definition (Document Paths). We define $P_{d_i} = \{p_{d_i}\}$ as the set of all existing paths in a document d_i: absolute paths and partial paths. We give a formal and recursive definition of P_{d_i} starting from the value v_{d_i} of document d_i.

For $v_{d_i} = \{a_{d_i,1} : v_{d_i,1}, \ldots, a_{d_i,n_i} : v_{d_i,n_i}\}$

- if $v_{d_i,j}$ is atomic: $P_{d_i} = P_{d_i} \cup S_{v_{d_i,j}}$;
- if $v_{d_i,j}$ is an object: $P_{d_i} = P_{d_i} \cup S_{v_{d_i,j}} \cup P_{v_{d_i,j}}$ where $P_{v_{d_i,j}}$ is the set of existing paths for the value $v_{d_i,j}$ (document paths for $v_{d_i,j}$);
- if $v_{d_i,j}$ is an array: $P_{d_i} = P_{d_i} \cup S_{v_{d_i,j}} \cup (\cup_{k=1}^{n_l} P_{v_{d_i,j,k}})$ where $P_{v_{d_i,j,k}}$ is the set of existing paths of the kth value of $v_{d_i,j}$ (document paths for $v_{d_i,j}$).

Since sets contain paths, the union of sets must be interpreted as a union of different paths. For example $\{a.b, a.b.c, a.b.d\} \cup \{a.b, b.a\} = \{a.b, a.b.c, a.b.d, b.a\}$.

Example. The document paths P_b for document (b) in Fig. 1 is as follows: $P_b =$

```
{_id,                                  director.first_name,
title,                                 director.last_name,
info,                                  first_name,
info.year,                             last_name,
year,                                  info.people.lead_actor,
info.country,                          info.people.lead_actor.first_name,
country,                               info.people.lead_actor.last_name,
info.link,                             people.lead_actor,
link,                                  people.lead_actor.first_name,
info.genres,                           people.lead_actor.last_name,
info.genres.1,                         lead_actor,
info.genres.2,                         lead_actor.first_name,
                                       lead_actor.last_name,
                                       info.people.actors,
                                       info.people.actors.1,
info.genres.3,                         info.people.actors.2,
genres,                                info.people.actors.3,
genres.1,                              people.actors,
genres.2,                              people.actors.1,
genres.3                               people.actors.2,
info.people,                           people.actors.3,
info.people.director,                  actors.1,
info.people.director.first_name,       actors.2,
info.people.director.last_name,        actors.3,
people                                 info.ranking,
people.director,                       info.ranking.score,
people.director.first_name,            ranking.score,
people.director.last_name,             score}
director,
```

Definition (Collection Paths). The set of all existing paths, absolute paths, partial paths and leaf nodes in a collection is called P_C and is defined as:

$$P_C = \cup_{i=1}^{n_c} P_{d_i}$$

We notice that $S_C \subseteq P_C$ (all absolute paths to any leaf node are included in P_C).

Definition (Dictionary). The dictionary $dict_C$ of a collection C is defined as:

$$dict_C = \{(p_k, \triangle_{p_k}^C)\}$$

where:

- $p_k \in P_C$ is an existing path in the collection C, $k \in [1..|P_C|]$;
- $\triangle_{p_k}^C = \{p_{k,1}, \ldots, p_{k,n_k}\} \subseteq S_C$ is the set of all *absolute paths* of the collection leading to p_k, $n_k = |\triangle_{p_k}^C|$.

Formally, the dictionary value $\triangle_{p_k}^C$ is a set of all absolute paths $p_{k,j} \in S_C$, $j \in [1..n_k]$, of the form $p_{k,j} = p_l.p_k$ where p_l is a path or p_l is empty. Thus, the dictionary value $\triangle_{p_k}^C$ contains all the absolute paths to p_k that exist in at least one document in the collection. For example, if we build the dictionary for a collection composed of document (b), the dictionary keys will contain *title* and *info.people*, but also *info.people.director*, *people.director*, *people*, *director* and so on.

Example. In the following example we present some dictionary entries from the collection (C) in Fig. 1:

- score:
  [ranking.score,
  others.ranking.score,
  info.ranking.score,
  classification.ranking.score]
- director:
  [director,
  film.details.director,
  info.people.director,

          description.director]
- lead_actor.first_name:
  [lead_actor.first_name,
  film.personas.lead_actor.
  first_name,
  info.people.lead_actor.
  first_name,
  description.stars.lead_actor.
  first_name]

4.4. Minimum closed kernel of operators

In this section we define a minimum closed kernel for operators based on the document operators defined in [34].

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

$$K = \{\sigma, \pi, \gamma, \mu, \lambda\}$$

The select, also called restriction (σ), the project (π), the aggregate (γ) and the unnest (μ) are unary operators whereas the lookup (λ) is a binary operator.

Definition (Query). If we take into consideration the kernel K for operators, a query Q is formulated by combining the previously presented unary and binary operators as follows:

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

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

We define the kernel as closed because each operator in the kernel operates across a collection and as a result, returns a new collection. Furthermore, we can observe that these operators are neither distributive, commutative nor associative; such operator combinations are valid in very particular cases only. Some algebraic manipulations are helpful in reducing the query complexity and they will be the subject of our future research work.

In our preliminary work the kernel was limited to project and select operators [21]. In recent work we have extended it to offer support for aggregate operators [31]. In this paper we

support two additional operators: unnest and lookup. Moreover, we give new definitions for the previously supported operators by adding additional features for the project and select operators as introduced in [34].

In the next sections, each operator is studied in five steps. We first give the operator definition, including partial paths. Next we give a query example for the operator and its evaluation in classical engines. We then explain how existing engines classically evaluate the operator. Finally, we define the operator reformulation rules which are illustrated with some reformulation examples. Defining the classical evaluation of operators is necessary in order to define the reformulation of operators so that these reformulations are evaluated in the same way as classical operators, particularly when considering missing paths and *null* values.

4.4.1. Selection

Definition (Selection). The select operator is defined as:

$$\sigma_P C_{in} = C_{out}$$

The select operator (σ) is a unary operator that filters the documents from collection C_{in} in order to retrieve only those that match the specified condition P. This can be a boolean combination expressed by the logical connectors $\{\vee, \wedge, \neg\}$ of atomic conditions, also called predicates, or a path check operation. The documents in C_{out} have the same document structures as the documents in collection C_{in}. However, the condition P may reduce the number of documents in C_{out} when applied to collection C_{in}.

The condition P is defined by a boolean combination of a set of triplets ($p_k \ \omega_k \ v_k$) where $p_k \subseteq P_{C_{in}}$ is a *path*, $\omega_k \in \{=; >; <; \neq; \geq; \leq\}$ is a comparison operator, and v_k is a value that can be atomic or complex. In the case of an atomic value, the triplet represents an atomic condition. In the case of a complex value, v_k is defined in the same way as a *document value* as defined in Section 4.2, $v_k = \{a_{k,1} : v_{k,1}, \ldots, a_{k,n} : v_{k,n}\}$ and ω_k is always "=". In this case the triplet represents a path check operation. We assume that each condition P is normalised to a conjunctive normal form:

$$P = \bigwedge\left(\bigvee p_k \ \omega_k \ v_k\right)$$

where $k \in [1..|P|]$, $|P|$ is the number of triplets in the select condition

Example. Let us suppose that we want to execute the following select operator on collection (C) from Fig. 1:

$$\sigma_{\text{year} \geq 2004 \wedge \text{director} = \{\text{"first\_name": "Clint", "last\_name":"Eastwood"}\})}(C)$$

Classical selection evaluation. During a selection evaluation, classical query engines return documents $d_i \in C_{in}$ based on the evaluation of the predicates $p_k \ \omega_k \ v_k$ of $P = \wedge(\vee \ p_k \ \omega_k \ v_k)$ as follows:

- if $p_k \in S_{d_i}$ the result of the predicate is *True/False* depending on the evaluation of $p_k \ \omega_k \ v_k$ in d_i;
- if $p_k \notin S_{d_i}$, the evaluation of $p_k \ \omega_k \ v_k$ is *False*.

The select operator will select only documents $d_i \in C_{in}$ where the evaluation of the normal form of condition P returns *True*.

Example. The previous selection operation only selects movies produced in *2004*, and the movie is directed by *Clint Eastwood* when the path *director* is an object with the following value *{"first_name": "Clint", "last_name":"Eastwood"}*. In a classical evaluation, the execution of this operation returns the following documents:

- {
  ```
  "_id":1,
  "title":"Million Dollar Baby",
  "year":2004,
  "genres":["Drama", "Sport"],
  "country":"USA",
  "director":{"first_name":"Clint", "last_name":"Eastwood"},
  "lead_actor":{"first_name":"Clint", "last_name":"Eastwood"},
  "actors":["Clint Eastwood", "Hilary Swank", "Morgan Freeman"],
  "ranking":{"score":8.1}}
  }
  ```

Because a classical evaluation only takes absolute paths into account, the result only contains the document (a) despite the presence of other documents (document (c)) which seem to satisfy the selection condition.

Selection reformulation rules. The reformulation of the select operator aims to filter documents based on a set of conditions from a collection of documents regardless of their underlying structures. The predicate triplets of the select condition are built across one path (atomic condition or path check). In practical terms, the query reformulation engine replaces each path used in a condition by all their corresponding absolute paths extracted from the dictionary. Therefore, a triplet condition $p_k \ \omega_k \ v_k$, $p_k \in P_{C_{in}}$ becomes a boolean "OR" combination of triplet conditions based on paths found in the dictionary. If we take into consideration the classical evaluation as defined above, the evaluation of this generated boolean "OR" combination in the reformulated select operator ensures that (i) a document containing at least one path can match the triplet condition, and (ii) a document containing no path evaluates the triplet condition as *False*.

$$\sigma_{P_{ext}}(C_{in}) = C_{out}$$

The query reformulation engine reformulates the normal form of predicates $P = \bigwedge\left(\bigvee p_k \ \omega_k \ v_k\right)$ by transforming each triplet $(p_k \ \omega_k \ v_k)$ into a disjunction of triplets, replacing the path p_k with the entries $\triangle_{p_k}^{C_{in}}$ while keeping the same operator ω_k and the same value v_k as follows : $(\bigvee_{\forall p_j \in \triangle_{p_k}^{C_{in}}} p_j \ \omega_k \ v_k)$. The reformulated normal form of the predicate is defined as:

$$P_{ext} = \bigwedge\left(\bigvee(\bigvee_{\forall p_j \in \triangle_{p_k}^{C_{in}}} p_j \ \omega_k \ v_k)\right)$$

Example. Let us suppose that we want to reformulate the select operator described above:

$$\sigma_{(\text{year} \geq 2004) \wedge (\text{director} = \{\text{"first\_name": "Clint", "last\_name":"Eastwood"}\})}(C)$$

The query reformulation engine reformulates each condition as follows:

- the condition $_{\text{year} \geq 2004}$ becomes:
  year ≥ 2004 ∨ info.year ≥ 2004 ∨ film.details.year ≥ 2004 ∨ description.year ≥ 2004

- the condition $_{\text{director} = \{\text{"first\_name":"Clint", "last\_name":"Eastwood"}\}}$ becomes:
  director={"first_name":"Clint", "last_name":"Eastwood"}
  ∨ info.people.director = {"first_name":"Clint", "last_name":"Eastwood"}
  ∨ film.details.director = {"first_name":"Clint", "last_name":"Eastwood"}
  ∨ description.director = "last_name":"Eastwood"}

After applying the reformulation rules, the select operator becomes:

$$\sigma_{(\text{year} \geq 2004 \vee \text{info.year} \geq 2004 \vee \text{film.details.year} \geq 2004 \vee \text{description.year} \geq 2004)}$$
$$\wedge (\text{director} = \{\text{"first\_name":"Clint", "last\_name":"Eastwood"}\} \vee \text{info.people.}$$
$$\text{director} = \{\text{"first\_name":"Clint", "last\_name":"Eastwood"}\}$$
$$\vee \text{film.details.director} = \{\text{"first\_name":"Clint", "last\_name":"Eastwood"}\}$$
$$\vee \text{description.director} = \{\text{"first\_name":"Clint", "last\_name":"Eastwood"}\})(C)$$

The execution of this latest select operator returns:

- ```
  {
   "_id":1,
   "title":"Million Dollar Baby",
   "genres":["Drama","Sport"],
   "country":"USA",
   "director":{
     "first_name":"Clint",
     "last_name":"Eastwood"},
   "lead_actor":{
     "first_name":"Clint",
     "last_name":"Eastwood"},
   "actors":["Clint Eastwood",
     "Hilary Swank","Morgan
     Freeman"],
   "ranking":{"score":8.1}}
  }
  ```

- ```
  {
   "_id":3,
  ```

```
"film":{
 "title":"Gran Torino",
 "details":{
   "year":2008,
   "country":"USA",
   "genres":"Drama",
   "director":{
     "first_name":"Clint",
     "last_name":"Eastwood"},
   "personas":{
     "lead_actor":{
       "first_name":"Clint",
       "last_name":"Eastwood"},
     "actors":["Clint Eastwood",
       "Bee Vang","Christopher
       Carley"]}},
   "others":{"ranking":{
       "score":8.1}}
 }
}
```

Executing the select operator after reformulation gives all the desired results, since it contains all the absolute paths that lead to the different selection conditions.

4.4.2. Projection
Definition (projection). The project operator is defined as:

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

The project operator (π) is a unary operator that projects only a specific portion from each document of a collection, i.e., only information referring to paths given in the query. In document stores, this operator is applied to a collection C_{in} by possibly projecting existing paths from the input documents, renaming existing paths or adding new paths as defined by the sequence of elements E. This returns an output collection C_{out}. The result contains the same number of documents as the input collection while the schema of the documents is changed.

The sequence of project elements is defined as $E = e_1, \ldots, e_{n_E}, n_E = |E|$, where each element e_j is in one of the following forms:

- (i) p_j, a path existing in the input collections; $p_j \in P_{C_{in}}$ which enables the projection of existing paths. As a result, the schema of the collection C_{out} may contain p_j;

- (ii) $p'_j : p_j$, where p'_j represents an absolute path (string in *Unicode* $\mathbb{A}^*$) to be injected into the structure of the collection C_{out} and p_j is an existing path in the input collection; $p_j \in P_{C_{in}}$ and its value is assigned to the new absolute path p'_j in C_{out}. This form renames the path p_j to p'_j in C_{out};

- (iii) $p'_j : [p_1, \ldots, p_m]$, where $[p_1, \ldots, p_m]$ is an array composed of m paths where $\forall l \in [1..m]$ $p_l \in P_{C_{in}}$ produces a new absolute path p'_j in C_{out} whose value is an array composed of the values obtained through the paths p_l;

- (iv) $p'_j : \beta$, where β is a boolean expression that compares the values of two paths in C_{in}, i.e., $\beta = (p_a \omega p_b)$, $p_a \in P_{C_{in}}$, $p_b \in P_{C_{in}}$ and $\omega \in \{=; >; <; \neq; \geq; \leq\}$. The evaluation of the boolean expression is assigned to the new absolute path p'_j in C_{out}.

Example. Let us suppose that we want to run the following project operator on collection (C) from Fig. 1:

$$\pi_{\substack{cond:director.first\_name = lead\_actor.first\_name, \\ desc:[title, genres], production\_year:year, ranking.score}}(C)$$

Classical projection evaluation. During a projection operation, classical query engines deal with missing paths or null values in the following documents with regards to the four possible forms of the projection element e_j:

- (i) p_j and (ii) $p'_j : p_j$ where p_j is a path from the input collection, $p_j \in S_{C_{in}}$:
  - If the path p_j leads to a value $v_{p_j} = null/atomic/object/array$ in a document $d_i \in C_{in}$, the corresponding document in the output collection $d'_i \in C_{out}$ contains: (i) the path p_j with the value v_{p_j} from d_i ($p_j \in S_{d'_i}$), (ii) the path p'_j with the value v_{p_j} from d_i
  - If the path $p_j \notin S_{d_i}$, where $d_i \in C_{in}$, the corresponding document in the output collection $d'_i \in C_{out}$ does not contain: (i) the path p_j, ($p_j \notin S_{d'_i}$), and (ii) the path p'_j ($p'_j \notin S_{d'_i}$);

- (ii) $p'_j : [p_1, \ldots, p_m]$ where $[p_1, \ldots, p_m]$ is an array of paths from the input collection and each $p_l \in S_{C_{in}}$. For a document $d_i \in C_{in}$, if the corresponding document in the output collection $d'_i \in C_{out}$ contains the path p'_j leading to an array that contains m values and one value for each p_l in $[p_1, \ldots, p_m]$, then the lth value is as follows:
  - If the path p_l leads to a value $v_{p_l} = null/atomic/object/array$ in the document d_i, the corresponding value is v_{p_l},
  - If the path $p_l \notin S_{d_i}$, the corresponding value is *null*;

- (iii) $p'_j : \beta$. β is the boolean expression $\beta = (p_a \omega p_b)$ where $p_a \in S_{C_{in}}$ and $p_b \in S_{C_{in}}$. For a document $d_i \in C_{in}$, then the corresponding document in the output collection $d'_i \in C_{out}$ contains the path p'_j leading to a boolean value:
  - If $p_a \in S_{d_i}$ and $p_b \in S_{d_i}$, the value is the boolean evaluation of β, *True/False*,
  - If $p_a \notin S_{d_i}$ and $p_b \in S_{d_i}$, the value is *False*,
  - If $p_a \in S_{d_i}$ and $p_b \notin S_{d_i}$, the value is *False*,
  - If $p_a \notin S_{d_i}$ and $p_b \notin S_{d_i}$, the value is *True*.

Example. The previous projection operation returns documents composed of the following paths:

- *cond*: the evaluation of a boolean expression which checks if the value of the path *director.first_name* is equal to the value of the path *lead_actor.first_name* or not, i.e., it checks whether both director and lead actor have the same first name or not;

- *desc*: an array composed of information from the *title* and *genre* paths;

- *production_year*: information from the path *year* using a new path called *production_year*, i.e., the path *year* from the input collection is renamed *production_year*;

- *ranking.score*: information from the path *ranking.score*, i.e., the same path as defined in the input collection is retained.

In a classical evaluation, the execution of this operation returns the following documents:

- ```
  {
   "_id":1,
   "cond":true,
   "desc": ["Million Dollar Baby",
     "Drama","Sport"],
   "production_year":2004,
   "ranking":{"score":8.1}
  }
  ```

- ```
  {
   "_id":2,
   "cond":true,
   "desc":["In the line of Fire",
     null]
  ```

```
}
```

- ```
  {
   "_id":3,
   "cond":true,
   "desc":[null,null]
  }
  ```

- ```
  {
   "_id":4,
   "cond":true,
   "desc":[null,null]
  }
  ```

The execution of the project operator gives rise to misleading results. We can see that only the first results include all the desired information. In the second result, only the *title* information is present for the new array *desc*.

We can see that in some cases the result which is *true* is not always real (case of document (*d*)) due to unreachable paths in the documents.

Projection reformulation rules. The aim of reformulating the project operator is to extract information from a collection of documents regardless of their underlying structures. In practical terms, the query reformulation engine replaces each path in the projection operation by their corresponding absolute paths extracted from the dictionary. In order to ensure that the reformulated operator has the same behaviour as the standard execution of the classical projection operation we introduce two specific notations, i.e., "|" and "‖" to deal with missing paths and *null* values.

In the operation $\pi_E(C_{in}) = C_{out}$, the original set of project elements E is extended as follows:

$E_{ext} = e_{1_{ext}}, \ldots, e_{n_{ext}}$ where each $e_{j_{ext}}$ is the extension of the $e_j \in E$. The extended project operator is defined as follows:

$$\pi_{E_{ext}}(C_{in}) = C_{out}$$

We introduce the notation "|" to exclude path p_j from the result when the project element e_j is atomic or the path p'_j if e_j is complex. In practical terms, an expression such as $p_{k,1}|\ldots|p_{k,n_j}$ is evaluated as follows for a document d_i:

- if $\exists p_k \in [p_{k,1}..p_{j,n_k}]$, where $p_k \in S_{d_i}$, then the corresponding document in the output collection $d'_i \in C_{out}$ contains the path p_k with the value v_{p_k} (from d_i);

- if $\nexists p_k \in [p_{k,1}..p_{k,n_k}]$, where $p_k \in S_{d_i}$, i.e., no path from the list is found in the document d_i, the corresponding document in the output collection $d'_i \in C_{out}$ does not contain the path p_k.

In the notation "|", if a first path from the list is found in the document, the corresponding value is kept for the output. Otherwise, the desired path is excluded from the output. Therefore, in the event where multiple paths are found in the document, the notation selects only the first one.

The notation "‖" is very similar to "|" notation when evaluating an expression such as $p_{k,1}\|\ldots\|p_{k,n_k}$ but it returns *null* instead of erasing the path in the output. It returns a *null* value in the following case:

- if $\nexists p_k \in [p_{k,1}..p_{k,n_k}]$, where $p_k \in S_{d_i}$, i.e., no path from the list is found in the document d_i, the operator returns a *null* value.

We can now define the following set of rules to extend each element $e_j \in E$ based on its four possible forms:

- (i) e_j is a path p_j in the input collection $p_j \in P_{C_{in}}, e_{j_{ext}} = p_{j,1} | \cdots | p_{j,n_j} \forall p_{j,k} \in \triangle_{p_j}^{C_{in}}$;

- (ii) $p'_j : p_j$, where p_j is a path, $p_j \in P_{C_{in}}$, then $e_{j_{ext}}$ is of the form $p'_j : p_{j,1} | \cdots | p_{j,n_j}, \forall p_{j,k} \in \triangle_{p_j}^{C_{in}}$;

- (iii) $p'_j : [p_1, \ldots, p_m]$, where $[p_1, \cdots, p_m]$ is an array of paths, then each path $p_j \in [p_1, \cdots, p_m]$ is replaced by a "‖" combination and $e_{j_{ext}}$ is of the form $p'_j : \left[p_{1,1} \| \cdots \| p_{1,n_1}, \ldots, p_{m,1} \| \cdots \| p_{m,n_m} \right] \forall p_{j,l} \in \triangle_{p_l}^{C_{in}}$;

- (iv) $p'_j : \beta$, where β is the boolean expression β, $e_{j_{ext}} = (p'_a \, \omega \, p'_b)$ where $p'_a = p_{a,1} | \cdots | p_{a,n_a}, \forall p_{a,l} \in \triangle_{p_a}^{C_{in}}$ and $p'_b = p_{b,1} | \cdots | p_{b,n_b}, \forall p_{b,l} \in \triangle_{p_b}^{C_{in}}$.

Example. Let us suppose that we want to reformulate the project operator described above:

$$\pi_{\substack{\text{cond:director.first\_name} = \text{lead\_actor.first\_name,} \\ \text{desc:[title, genres], production\_year: year, ranking.score}}}(C)$$

Below we present the results of applying the reformulation rules to each element of the project operator:

- the element $\text{cond:director.first\_name} = \text{lead\_actor.first\_name}$ becomes:

  $\text{cond:} p'_a = p'_b$

  where

  $p'_a = \text{director.first\_name} \mid \text{info.people.director.first\_name} \mid$ film.details.director.first_name $\mid$ description.director.first_name

  $p'_b = \text{lead\_actor.first\_name} \mid \text{info.people.lead\_actor.first\_name} \mid$ film.details.personas.lead_actor.first_name $\mid$ description.stars.lead_actor.first_name

- the element $\text{desc:[title, genres]}$ becomes:

  $\text{desc:}[p'_1, p'_2]$

  where

  $p'_1 = \text{title} \| \text{film.title} \| \text{description.title}$

  $p'_2 = \text{genres} \| \text{info.genres} \| \text{film.details.genres} \|$ classification.genres

- the element $\text{production\_year:year}$ becomes:

  production_year: year $\mid$ info.year $\mid$ film.details.year $\mid$ description.year

- the element ranking.score becomes:

  ranking.score $\mid$ info.ranking.score $\mid$ film.others.ranking.score $\mid$ classification.ranking.score

After applying the reformulation rules, and with reference to previous paragraphs for reformulations, the project operator becomes:

$$\pi_{\substack{\text{cond:} p'_a = p'_b, \text{ desc:}[p'_1, p'_2], \text{ production\_year:year} \mid \\ \text{info.year} \mid \text{film.details.year} \mid \text{description.year, ranking. score} \mid \text{info.ranking.score} \mid \\ \text{film.others.ranking.score} \mid \text{classification.ranking.score}}}(C)$$

The execution of this latest project operator returns:

- ```
  {
  "_id":1,
  "ranking":{"score":8.1},
  "cond":true,
  "desc":['Million Dollar Baby',
     'Clint'],
  "production_year":2004
  }
  ```

- ```
  {
  "_id":2,
  "info":{"ranking":
     {"score":7.2}},
  "cond":true,
  "desc":[ 'In the Line of Fire',
     'Clint'],
  "production_year":1993
  }
  ```

- ```
  {
  "_id":3,
  "film":{"others":
     {"ranking":
        {"score":8.1}}},
  "cond":true,
  "desc":[ 'Gran Torino',
     'Clint'],
  "production_year":2008
  }
  ```

- ```
  {
  "_id":4,
  "classification":
     {"ranking":
        {"score":7.2}},
  "cond":false,
  "desc":[ 'The Good, the
     Bad and the Ugly',
     'Clint'],
  "production_year":1966
  }
  ```

The reformulated project operator is now able to reach all the paths from the initial query regardless of their numerous locations inside the collection. In addition, the comparison of path information now gives reliable results.

4.4.3. Aggregation
Definition (Aggregation). The aggregate operator is defined as:

$$_G\gamma_F(C_{in}) = C_{out}$$

The aggregate operator is a unary (γ) operator grouping documents according to the values from the grouping conditions G. The output is a collection of documents where each document refers to a group and contains a computed aggregated value as defined by the aggregation function F.

- G represents the grouping conditions, $G = p_1, \ldots, p_g$, where $\forall k \in [1..g]$, $p_k \in P_{C_{in}}$;
- F is the aggregation function, $F = p : f(p_f)$, where p represents the new path in C_{out} for the value computed by the aggregation function f for the values reached by the path p_f where $p_f \in P_{C_{in}} \wedge p_f \notin G$, $f \in \{Sum, Max, Min, Avg, Count\}$.

Example. Let us suppose that we want to run the following aggregation operation on collection (C) from Fig. 1:

$$_{ranking.score}\gamma_{titles\_count:Count(title)}(C)$$

Classical aggregation evaluation. During an aggregation evaluation, classical query engines perform as follows based on the paths in $G = p_1, \ldots, p_g, p_i \in S_{C_{in}}$ and p_f ($F = p : f(p_f)$, $p_f \in S_{C_{in}}$):

- In the grouping step, documents are grouped according to the presence or non-presence of the paths from $G = p_1, \ldots, p_g$ in documents. Documents are grouped when they have the same subset of paths from G and the same values for these paths. Finally, a group is created for those documents that contain no paths from G. Formally, a group is a subset of documents $\{d\}$ such that: (i) $\exists H = h_1, \ldots, h_h$, $\forall i \; h_i \in G$ or H is empty, (ii) $\forall d$ document of the group, $\forall h_i \in H$, $h_i \in S_d$ and every d has the same value $\forall i \; h_i \in H$;
- In the computation step, for each group established in the grouping step, the function f is applied as follows:
  - If $\exists \; d$ in the group is such that $p_f \in S_d$, then f is computed across all documents d_i of the group where $p_f \in S_{d_i}$, and documents d_k of the group where $p_f \notin S_{d_k}$ are simply ignored,
  - If $\nexists \; d$, a document from the group, is such that $p_f \in S_d$, then f is evaluated as a *null* value (no matter what f is).

Example. The previous aggregation operation groups movies by their scores as defined in the path *ranking.score* and counts the number of titles (movies) for each group.
The native query engine returns the following results:

- ```
  {
    "_id":null,
    "titles_counts":3
  }
  ```
- ```
  {
    "_id":8.1,
    "titles_counts":1
  }
  ```

These results place document (*a*) with a *ranking.score* of *8.1* in one group and the other documents (*b, c, d*) in a second group with a *ranking.score* of *null* since this path is unreachable in the remaining documents.

Aggregation reformulation rules. The aim of reformulating the aggregate operator is to replace each path from the grouping and aggregation function by their corresponding absolute paths extracted from the dictionary. Nevertheless, a preliminary project operation is needed to unify the heterogeneous paths in documents with a set of common paths for all documents. Then a classical aggregation is applied to the previously projected documents. In practical terms, the query reformulation engine first starts by projecting out all values reached by the paths from both G (grouping conditions) and F (aggregation function). This project operation renames the distinct absolute paths extracted from the dictionary for paths in G ($G = p_1, \ldots, p_g$) and F (path p_f) to the paths initially expressed in the original query.

Then we apply the classical aggregate operator to the output of the added project operator.

Let Att be the set of all paths expressed in G and F, that is $Att = G \cup \{p_f\}$. The additional project operator is defined as:

$$\pi_{E_{ext}}(C_{in})$$

where $E_{ext} = \cup_{\forall p_j \in Att}\{p_j : p_{j,1}|\ldots|p_{j,n_j}\}, \forall p_{j,k} \in \triangle_{p_j}^{C_{in}}$

The reformulated aggregate operator is formally defined as:

$$_G\gamma_F(\pi_{E_{ext}}(C_{in})) = C_{out}$$

Example. Let us suppose that we want to reformulate the aggregate operator as described above:

$$_{(ranking.score)}\gamma_{(titles\_count: \; Count(title))}$$

To reformulate the aggregate operator, the query reformulation engine must first generate a project operator, which is defined as follows:

$$\pi_{\substack{ranking.score:ranking.score \; | \; info.ranking.score \\ | \; others.ranking.score \; | \; classification.ranking.score, \\ title:title \; | \; film.title \; | \; description.title}}(C)$$

The aggregate operator after reformulation becomes:

$$_{ranking.score}\gamma_{titles\_count:Count(\; title)} \Big(\pi_{\substack{ranking.score:ranking.score \; | \; info.ranking.score \\ | \; others.ranking.score \; | \; classification.ranking.score, \\ title:title \; | \; film.title \; | \; description.title}}(C) \Big)$$

Now after executing this query we obtain the following results:

- ```
  {
    "_id":7.2,
    "titles_count":2
  }
  ```
- ```
  {
    "_id":8.1,
    "titles_count":2
  }
  ```

4.4.4. Unnest
Definition (Unnest). The unnest operator is defined as:

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

The unnest operator (μ) is a unary operator which flattens an array reached via a path p in C_{in}. For each document $d_i \in C_{in}$ that contains p, the unnest operator outputs a new document for each element of the array. The structure of the output documents is identical to the original document d_i, except that p (initially an array) is replaced by a path leading to one value of the array in d_i.

Example. Let us suppose that we want to run the following unnest operation on collection (C) from Fig. 1:

$\mu_{genres}(C)$

Classical unnest evaluation. During an unnest evaluation, classical query engines generate new documents for the operation $\mu_p(C_{in}) = C_{out}$ as follows:

- If $p \in S_{d_i}$, the collection C_{out} contains new k documents where $k = |v_p|$ is the number of entries of the array referenced by the path p. Each new document contains the path p. The value of p in each new document $d_{i,j}$ is equal to the jth entry from the array value v_p in d_i;
- If $p \notin S_{d_i}$, the collection C_{out} contains a copy of the original document d_i.

Example. The previous unnest operator takes into account the array referenced by the path *genres* and returns a new document for each element in the array. By executing this query, the unnest operator only applies to document (*a*) due to the presence of the absolute path *genres* in this document. As a result, the array *genres* from document (*a*) is split into two documents as follows:

- ```
  {
    "_id":1,
    "title":"Million Dollar
        Baby",
    "year":2004,
    "genres":"Drama",
    "country":"USA",
    "director":{
        "first_name":"Clint",
        "last_name":"Eastwood"},
    "lead_actor":{
        "first_name":"Clint",
        "last_name":"Eastwood"},
    "actors":[ "Clint Eastwood",
        "Hilary Swank", "Morgan
        Freeman"],
    "ranking":{"score":8.1}
  }
  ```
- ```
  {
    "_id":1,
    "title":"Million Dollar
        Baby",
    "year":2004,
    "genres":"Sport",
    "country":"USA",
    "director":{
        "first_name":"Clint",
        "last_name":"Eastwood"},
    "lead_actor":{
        "first_name":"Clint",
        "last_name":"Eastwood"},
    "actors":["Clint Eastwood",
        "Hilary Swank", "Morgan
        Freeman"],
    "ranking":{"score":8.1}
  }
  ```

Unnest reformulation rules. The aim of reformulating the unnest operator is to generate documents where on each occasion the path p contains an element from the initial array referenced by p in the collection C_{in} regardless of the underlying structure of the documents. In practical terms, the query reformulation engine combines a series of different unnest operators applied to each path p_j extracted from the dictionary entry $\triangle_p^{C_{in}}$ that leads to the path p. We represent the combination of the operators by using the " $\circ$ " composition symbol. The reformulation of the unnest operator is formally defined as:

$$\circ_{\forall p_j} \in \triangle_p^{C_{in}} \mu_{p_j}(C_{in})$$

Example. Let us suppose that we want to reformulate the following unnest operation as described above:

$\mu_{genres}(C)$

After applying the above-mentioned transformation rules, the unnest operation becomes:

$$\mu_{genres} \circ \mu_{info.genres} \circ \mu_{film.details.genres} \circ \mu_{classification.genres}(C)$$

Now, executing this query returns seven documents where the array from document (a) generates two documents which have the same information as document (a) and the array becomes a simple attribute whose value is an entry from the array. We

obtain three documents from document (b) (the array *genres* contains three entries). Document (c) stays invariant. Finally, document (d) returns one document (the array *genres* contains only a single entry):

- ```
  {
    "_id":1,
    "title":"Million Dollar
        Baby",
    "year":2004,
    "genres":"Drama",
    "country":"USA",
    "director":{
        "first_name":"Clint",
        "last_name":"Eastwood"},
    "lead_actor":{
        "first_name":"Clint",
        "last_name":"Eastwood"},
    "actors":["Clint Eastwood",
        "Hilary Swank",
        "Morgan Freeman"],
    "ranking":{"score":8.1}
  }
  ```
- ```
  {
    "_id":1,
    "title":"Million Dollar Baby",
    "year":2004,
    "genres":"Sport",
    "country":"USA",
    "director":{
        "first_name":"Clint",
        "last_name":"Eastwood"},
    "lead_actor":{
        "first_name":"Clint",
        "last_name":"Eastwood"},
    "actors":["Clint Eastwood",
        "Hilary Swank", "Morgan
        Freeman"],
    "ranking":{"score":8.1}
  }
  ```
- ```
  {
    "_id":2,
    "title":"In the Line of Fire",
    "info":{
        "year":1993,
        "country":"USA",
        "genres":"Drama",
        "people":{
        "director":{
            "first_name":"Clint",
            "last_name":"Eastwood"},
        "lead_actor":{
            "first_name":"Clint",
            "last_name":"Eastwood"},
        "actors":["Clint Eastwood",
        "John Malkovich", "Rene
        Russo Swank"]
        },
        "ranking":{"score":7.2}
    }
  }
  ```
- ```
  {
    "_id":2,
    "title":"In the Line of Fire",
    "info":{
        "year":1993,
  ```
- ```
        "country":"USA",
        "genres":"Action",
        "people":{
        "director":{
            "first_name":"Clint",
            "last_name":"Eastwood"},
        "lead_actor":{
            "first_name":"Clint",
            "last_name":"Eastwood"},
        "actors":["Clint Eastwood",
        "John Malkovich",
        "Rene Russo Swank"]
        },
        "ranking":{"score":7.2}
    }
  }
  ```
- ```
  {
    "_id":2,
    "title":"In the Line of Fire",
    "info":{
        "year":1993,
        "country":"USA",
        "genres":"Crime",
        "people":{
        "director":{
            "first_name":"Clint",
            "last_name":"Eastwood"},
        "lead_actor":{
            "first_name":"Clint",
            "last_name":"Eastwood"},
        "actors":[ "Clint Eastwood",
        "John Malkovich",
        "Rene Russo Swank"]
        },
        "ranking":{"score":7.2}
    }
  }
  ```
- ```
  {
    "_id":3,
    "film":{
    "title":"Gran Torino",
    "details":{
    "year":2008,
    "country":"USA",
    "genres":"Drama",
    "director":{
        "first_name":"Clint",
        "last_name":"Eastwood"},
    "personas":{
        "lead_actor":{
        "first_name":"Clint",
        "last_name":"Eastwood"},
        "actors":["Clint Eastwood",
        "Bee Vang", "Christopher
        Carley"]
    }
    },
    "others":{
        "ranking":{"score":8.1}
    }
    }
  }
  ```

4.4.5. Lookup

Definition (Lookup). The lookup operator is defined as: The lookup operator (λ) is a binary operator which enriches (embeds or left-

joins) documents from the input collection C_{in} with documents from the external collection C_{ex} that satisfy a lookup condition. This condition determines whether the values of paths reached from local paths p_{in} in C_{in} match the values reached via external paths p_{ex} in C_{ex} or not. This operator is similar to the *left outer join* operator in relational algebra. As a result, the lookup operator adds an array *res* to each document from C_{in} and each element of *res* is a document from C_{ex} that satisfies the lookup condition $p_{in} = p_{ex}$. The output collection C_{out} is the same size as the input collection C_{in}. The structure of the documents in C_{out} are slightly different from C_{in} because each document in C_{out} includes an additional path *res* whose value is an array of the nested external documents. Despite lookup and unnest operators being used to nest or unnest values, it is important to underline that lookup and unnest operators are not reverse operators.

Example. Let us suppose that we want to run the following lookup operation on collection (C) from Fig. 1:

$$(C)\lambda_{\ dir\_actor:director.first\_name=lead\_actor.first\_name}(C)$$

Classical lookup evaluation. During a lookup evaluation, classical query engines deal with misleading paths or null values in documents based on the evaluation of the condition $p_{in} = p_{ex}$ as follows:

- If $p_{in} \in S_{d_i}$, $d_i \in C_{in}$, *res* contains an array with all documents $d_j \in C_{ex}$ where $p_{ex} \in S_{d_j}$ and $v_{p_{in}} = v_{p_{ex}}$;
- If $p_{in} \notin S_{d_i}$, $d_i \in C_{in}$, *res* contains an array with all documents $d_j \in C_{ex}$ where $p_{ex} \notin S_{d_j}$.

Example. The previous lookup operator left joins each film based on the director's first name with other films that have the same first name for the main actor. The execution of this query returns one entry in the new path *dir_actor* for document (a). This entry contains the information from document (a) since the lookup operation can only match the information from document (a). The content of the new path *dir_actor* for document (a) is as follows:

```
• "dir_actor":[                          "lead_actor":{
     {                                     "first_name":"Clint",
      "_id":1,                             "last_name":"Eastwood"},
      "title":"Million Dollar Baby",      "actors":[ "Clint Eastwood",
      "year":2004,                         "Hilary Swank",
      "genres":[ "Drama","Sport"],        "Morgan Freeman"],
      "country":"USA",                     "ranking":{"score":8.1}
      "director":{                        }
       "first_name":"Clint",            ]
       "last_name":"Eastwood"},
```

Here we explain the classical evaluation process and the possible incorrect results. The lookup succeeds in matching document (a) with itself, but despite the presence of other documents that may satisfy the lookup condition we can see that they are absent from the new path *dir_actor*. We can see this same result inside the remaining documents (b, c, d) that give three documents as a result, and each resulting document contains the same value for the new path *dir_actor*:

```
"dir_actor": [{                           "first_name":"Clint",
 "_id":2,                                 "last_name":"Eastwood"},
 "title":"In the Line of Fire",        "actors":["Clint Eastwood",
 "info":{                                 "Bee Vang",
 "year":1993,                             "Christopher Carley"]
 "country":"USA",                        }
 "genres":[ "Drama","Action",          }, "others":{"ranking":
 "Crime"],                                 {"score":8.1}
 "people":{                             }
  "director":{                         }
   "first_name":"Clint",             },
   "last_name":"Eastwood"},           {
  "lead_actor":{                        "_id":4,
   "first_name":"Clint",               "description":{
   "last_name":"Eastwood"},            "title":"The Good, the Bad
  "actors":[ "Clint Eastwood",           and the Ugly",
  "John Malkovich",                     "year":1966,
  "Rene Russo Swank"]                   "country":"Italy",
 },                                     "director":{
 "ranking":{"score":7.2}                 "first_name":"Sergio",
 }                                        "last_name":"Leone"},
},                                      "stars":{"lead_actor":{
{                                         "first_name":"Clint",
 "_id":3,                                 "last_name":"Eastwood"},
 "film":{                               "actors":[ "Clint Eastwood",
 "title":"Gran Torino",                 "Eli Wallach",
 "details":{                            "Lee Van Cleef"]
  "year":2008,                         }
  "country":"USA",                     },
  "genres":"Drama",                    "classification":{
  "director":{                         "ranking":{"score":7.2},
   "first_name":"Clint",               "genres":[ "Western"]
   "last_name":"Eastwood"},           }
  "personas":{                        }
  "lead_actor":{                     ]
```

We can see that this result does not contain the expected information, for instance, document (d) should not match any of the other documents since the *director.first_name* is totally different from the *lead_actor.first_name*. It is supposed to return an empty array for the new path *dir_actor*. Also, document (a) is excluded from the results. Therefore, the evaluation of the lookup condition returns *True* when both paths from the input collection and the output collection are not found in the documents. In this case, the operator performs join.

Lookup reformulation rules. The aim of reformulating the lookup operator is to replace each path from the join condition by their corresponding absolute paths extracted from the dictionaries. We reuse the previously defined notation "|" to ensure an identical evaluation for the reformulated lookup compared to the classical evaluation mentioned in the previous paragraph. We observe that the lookup reformulation requires a dictionary for the input collection C_{in} and for the external collections C_{ex}. In practical terms, the query reformulation engine includes a combination of all absolute paths of $\triangle_{p_{in}}^{C_{in}}$ and a combination of all absolute paths of $\triangle_{p_{ex}}^{C_{ex}}$. The reformulated lookup operation is defined as:

$$(C_{in})\lambda_{res:p_{j,1}|...|p_{j,n_j} = p_{l,1} | ... | p_{l,n_l}}(C_{ex}) = C_{out}$$
$$\forall p_{j,x} \in \triangle_{p_{in}}^{C_{in}}, \ \forall p_{l,y} \in \triangle_{p_{ex}}^{C_{ex}}$$

Example. Let us suppose that we want to reformulate the following lookup operation:

$$(C)\lambda_{dir\_actor:director.first\_name=lead\_actor.first\_name}(C)$$

Below is the reformulation of the lookup operation:

$$(C)\lambda_{dir\_actor:director.first\_name\ |info.people.director.first\_name}$$
$$|film.details.director.first\_name\ |\ description.director.first$$
$$\_name=lead\_actor.first\_name|\ info.people.lead\_actor.$$
$$first\_name\ |film.details.personas.lead\_actor.first\_name\ |$$
$$description.stars.lead\_actor.first\_name(C)$$

The execution of this lookup operation gives three documents, i.e., documents (a, b, c). Each resulting document contains the same value for the new path *dir_actor*:

```
• "dir_actor":[
  {
    "_id":1,
    "title":"Million Dollar Baby",
    "year":2004,
    "genres":["Drama", "Sport"],
    "country":"USA",
    "director":{
    "first_name":"Clint",
    "last_name":"Eastwood"},
    "lead_actor":{
    "first_name":"Clint",
    "last_name":"Eastwood"},
    "actors":[ "Clint Eastwood",
    "Hilary Swank",
    "Morgan Freeman"],
    "ranking":{"score":8.1}
  },
  {
    "_id":2,
    "title":"In the Line of Fire",
    "info":{
    "year":1993,
    "country":"USA",
    "genres":["Drama", "Action",
    "Crime"],
    "people":{
      "director":{
        "first_name":"Clint",
        "last_name":"Eastwood"},
      "lead_actor":{
        "first_name":"Clint",
```
```
    "last_name":"Eastwood"},
    "actors":[ "Clint Eastwood",
      "John Malkovich",
      "Rene Russo Swank"]
    },
    "ranking":{"score":7.2}
    }
  },
  {
    "_id":3,
    "film":{
    "title":"Gran Torino",
    "details":{
      "year":2008,
      "country":"USA",
      "genres":"Drama",
      "director":{
        "first_name":"Clint",
        "last_name":"Eastwood"},
      "personas":{
        "lead_actor":{
        "first_name":"Clint",
        "last_name":"Eastwood"},
      "actors":[ "Clint Eastwood",
        "Bee Vang",
        "Christopher Carley"]
      }
    },
    "others":{
        "ranking":{"score":8.1}
    }
    }
    }
]
```

However, the document (d) does not have the same information for the paths *director.first_name* and *lead_actor.first_name*. Therefore, the lookup operation returns the following result:

```
• "dir_actor":[]
```

4.5. Algorithm for automatic operator reformulation

In this section we introduce the query extension algorithm that automatically enriches the user query.

If we take into account the definition of a user query (Section 4.4), the goal of the extension Algorithm 1 is to modify the composition of the query in order to replace each operator by its extension (defined in the previous sections). The final extended query is then the composition of the reformulated operators corresponding to $q_1 \circ \cdots \circ q_r$.

Ultimately, the native query engine for document-oriented stores, such as MongoDB, can execute the reformulated queries. It is therefore easier for users to find all the desired information regardless of the structural heterogeneity inside the collection.

5. Experiments

In this section we present a series of experiments to evaluate *EasyQ* and validate its ability for enabling schema-independent querying for NoSQL document-oriented databases. We conducted all our experiments on an Intel I5 i5-4670 processor with a clock speed of 3.4 GHz and 4 cores per socket. The machine had 16 GB of RAM and a 1TB SSD drive. We ran all the experiments in single-threaded mode. We chose MongoDB as the underlying document store for our experiments. We focused on the measurement of the execution time for each executed query.

We implemented the Algorithm 1 using the Python programming language. We used PyMongo, which is a Python distribution containing tools for working with MongoDB. *EasyQ* takes as input the collection name, the query representing a combination of operators and the set of parameters. Using the reformulation algorithm, *EasyQ* automatically reformulates the input query and executes it directly on MongoDB. Results are automatically displayed to the user in the raw format as the MongoDB query engine returns them.

In addition to *EasyQ*, we developed a module to build the dictionary. The dictionary construction module runs a recursive algorithm that goes through all documents trees starting from the root down to each leaf node before going up to collect all the absolute paths, partial paths and leaf nodes. All documents in the collection are involved in this process.

The purpose of the experiments was to answer the following questions:

- What are the effects on the execution time of the rewritten queries when the size of the collection is varied and is this cost acceptable or not?
- Is the time to build the dictionary acceptable, and is the size of the dictionary affected by the number of structures in the collection?

5.1. Experimental protocol

In this section we describe the different customized synthetic datasets that we generated to run our experiments, (all datasets are available online[1]). Furthermore, we define the query set that we used to evaluate the *EasyQ* query reformulation engine.

5.1.1. Datasets

In order to analyse the behaviour of *EasyQ* on varying collection sizes and structures, we generated customized synthetic datasets. First we collected a CSV document with information relating to 5000 movies. Then we started generating an initial homogeneous dataset that we called *Baseline* where documents within the different collections are composed of 10 attributes (4 primitive type attributes, 3 array type attributes and 3 complex attributes of an object type in which we nested additional primitive attributes). All documents within the different collections in the *Baseline* dataset share the same structure as document (a) in Fig. 1. We intentionally chose to work with attributes that may be absent from some documents, e.g., *awards* in collection (C) in Fig. 1. Moreover, we may have some attributes with *null* values, e.g., *link*. Fig. 4(a) illustrates a document from the *Baseline* dataset. We used the *Baseline* dataset as baseline for our experiments. It helped us to compare our schema-independent querying mechanism with the normal execution of queries on collections that have a unique homogeneous structure. The *Baseline* dataset was composed of five collections of 1 M, 10 M, 25 M, 50 M, 100 M and 500 M documents for a total disk space ranging from 500 MB to more than 250 GB.

In addition to the *Baseline* dataset, we then injected heterogeneity into the structure of documents from the *Baseline* dataset. We opted to introduce structural heterogeneity by changing the location of the attributes of the documents from the *Baseline* dataset. We introduced new absolute paths with variable lengths.

---

[1] https://www.irit.fr/recherches/SIG/SDD/EASY-QUERY/

Algorithm 1: Automatic Query Reformulation Algorithm*

```
1  input:  Q                                                          // original query
2  output:  Q_ext                                                     // reformulated query
3  Q_ext ← id                                                         // identity
4  foreach q_i ∈ Q                                                    // for each operator in Q
5  do
6      switch q_i                                                     // case of the operator q_i
7      do
8          case π_E :                                                 // q_i is a project operator
9
10             E_ext ← ∅                                              // initialising the set of extended elements from E
11             foreach e_j ∈ E                                        // for each element e_j ∈ E
12             do
13                 if e_j = p_j is a path (p_j ∈ P_{C_in})            // e_j takes the form of a path
14                 then
15                     e_{j_ext} = p_{j,1} | … | p_{j,n_{n_j}} ∀p_{j,l} ∈ △^{C_in}_{p_j}   // generating p_{j_ext} using paths from △^{C_in}_{p_j}
16
17                 if e_j = p'_j : p_j, (p_j ∈ P_{C_in})              // renaming the path p_j to p'_j
18                 then
19                     e_{j_ext} = p'_j : p_{j,1} | … | p_{j,n_j}, ∀p_{j,l} ∈ △^{C_in}_{p_j}   // generating e_{j_ext} while renaming paths from △^{C_in}_{p_j} to p'_j
20
21                 if e_j = p'_j : [p_1, …, p_{m_j}], ∀l ∈ [1..m_j], p_l ∈ S_{C_in}   // new array [p_1, …, p_{m_j}] composed of paths p_l
22                 then
23                     e_{j_ext} = p'_j : [p_{1,1}|| …|| p_{1,n_1} , …, p_{m,1}|| …|| p_{m,n_m}] ∀p_{j,l} ∈ △^{C_in}_{p_l}
24                 if e_j = p'_j : β, β = (p_a ω p_b)                 // comparing values of paths p_a and p_b
25                 then
26                     e_{j_ext} = p_{a,1} | … | p_{a,n_a} ω p_{b,1} | … | p_{b,n_b}, ∀p_{a,k} ∈ △^{C_in}_{p_a}, ∀p_{b,l} ∈ △^{C_in}_{p_b}
27                 E_ext = E_ext ∪ {e_{j_ext}}                        // extending E_ext by the new extended element e_{j_ext}
28             end
29             Q_ext ← Q_ext ∘ π_{E_ext}                              // adding the extended projection π_{E_ext} to Q_ext
30         end
31         case σ_P :                                                 // q_i is a select operator and the condition is normalised to P = ⋀(⋁ p_k ω_k v_k)
32
33             P_ext ← ⋀(⋁(⋁_{∀p_j∈△^{C_in}_{p_k}} p_j ω_k v_k))     // extending the condition with a disjunction ⋁_{∀p_j∈△^{C_in}_{p_k}} p_j ω_k v_k
34
35             Q_ext ← Q_ext ∘ σ_{P_ext}                             // adding the extended selection σ_{P_ext} to Q_ext
36         end
37         case _Gγ_F :                                               // q_i is an aggregate operator
38         where G = p_1, …, p_g, and F = p : f(p_f)
39
40             E_ext ← ∅
41             foreach p_j ∈ {G} ∪ {p_f}                              // for each attribute in G and F
42             do
43                 E_ext = E_ext ∪ {p_j : p_{j,1}|…|p_{j,n_j}}, ∀p_{j,l} ∈ △^{C_in}_{p_j}   // generating elements where paths △^{C_in}_{p_j} are renamed to p_j
44             end
45             Q_ext ← Q_ext ∘ (_Gγ_F ∘ π_{E_ext})                   // adding the combined aggregation _Gγ_F and the custom projection π_{E_ext} to Q_ext
46         end
47         case μ_p :                                                 // q_i is an unnest operation
48
49             foreach p_j ∈ △^{C_in}_p                               // for each attribute p_j in △^{C_in}_p
50             do
51                 Q_ext ← Q_ext ∘ μ_{p_j}                            // extending Q_ext with μ_{p_j}
52             end
53         end
54         case λ_{res:p_in=p_ex} :                                  // q_i is a lookup operation
55
56             Q_ext ← Q_ext ∘ λ_{res:p_{j,1} | … | p_{j,n_j} = p_{l,1} | … | p_{l,n_l}} ∀p_{j,x} ∈ △^{C_in}_{p_in}, ∀p_{l,y} ∈ △^{C_ex}_{p_ex}
57     end
58     endsw
59 end
60 return Q_ext
```
61 * For the purpose of this paper, details regarding temporary dictionary maintenance are not presented in this algorithm

---

The process of generating the heterogeneous collection took several parameters into account: the number of structures, the depth of the absolute paths and the number of new attributes of an object type. We randomly nested a subset of attributes, e.g., up to 10 attributes, under these complex attributes at pre-defined depths. The complex attributes are unique in each structure, which enables unique absolute paths for each attribute in each structure.

Fig. 4(b) describes a sample of a generated document along with the parameters used to generate it. Therefore, for each attribute there are as many absolute paths as the chosen number of structures.

For the purpose of the experiments we used the above mentioned strategy to generate the following datasets:

- A *Heterogeneous* dataset to evaluate the execution time of the reformulated query on varying collection sizes. This dataset was composed of five collections of 1 M, 10 M, 25 M, 50 M, 100 M and 500 M documents for a total disk space ranging from 500 MB to more than 250 GB and each collection contained 10 schemas;

- A *Schemas* dataset to evaluate the time required to reformulate a query for a varying number of schemas and to study the consequences on the dictionary size. This dataset was composed of five collections of 100 M documents with *10, 100, 1000, 3000* and *5000* schemas respectively;

- A *Structures* dataset to evaluate the time required to execute a query for a varying number of schemas. This dataset was composed of five collections of 10 M documents with *10, 20, 50, 100* and *200* schemas respectively;

```
{ "_id":1,
  "title":"Million Dollar Baby",
  "year":2004,
  "link":null,
  "awards":["Oscar",  "Golden Globe",
     "Movies for Grownups Award", "AFI
     Award"],
  "genres":["Drama", "Sport"],
  "country":"USA",
  "director":{ "first_name":"Clint",
     "last_name":"Eastwood"
  },
  "lead_actor":{ "first_name":"Clint",
     "last_name":"Eastwood"
  },
  "actors":["Clint Eastwood",
     "Hilary Swank", "Morgan Freeman"],
  "ranking":{ "score":8.1
  }
}
```

(a) Document from the *Baseline* dataset

```
{"_id":1
 "group_1A":
   {"level0":
     {"level1":
       {"level2":
         {"level3":
           {"ranking" : {"score": 8.1},
            "country" : "USA",
            "lead_actor" : {"first_name": "Clint", "last_name": "Eastwood"},
            "director" : {"first_name": "Clint", "last_name": "Eastwood"},
            "link" : null
           }
         }
       }
     }
   },
 "group_2A":
   {"level0":
     {"level1":
       {"level2":
         {"level3":
           {"genres" : ["Clint Eastwood", "Hilary Swank", "Morgan Freeman"]}
         }
       }
     }
   },
 "group_3A":
   {"level0":
     {"level1":
       {"level2":
         {"level3":
           {"title" : "Million Dollar Baby",
            "year" : 2004,
            "actors":["Clint Eastwood", "Hilary Swank", "Morgan Freeman"],
            "awards":["Oscar",  "Golden Globe", "Movies for Grownups Award",
               "AFI Award"]
           }
         }
       }
     }
   }
}
```

(b) Document from the *Heterogeneous* dataset (3 groups, 5 nesting levels)

Fig. 4. Examples From the *Baseline* and the *Heterogeneous* dataset used in query reformulation evaluation.

Table 2
Settings of the *Heterogeneous* dataset for query reformulation evaluation.

Setting	Value
# of schemas	10
# of grouping objects per schema (width heterogeneity)	{5,6,1,3,4,2,7,2,1,3}
Nesting levels per schema (depth heterogeneity)	{4,2,6,1,5,7,2,8,3,4}
Avg. percentage of schema presence	10%
# of leaf nodes per schema	9 or 10
# of attributes per grouping objects	[1…10]

Table 3
Workloads query elements.

Path	Attribute	Type	Paths	Depths
p1	director.first_name	String	10	{3,6,5,4,8,9,5,7,2,3}
p2	lead_actor.first_name	String	10	{3,6,5,4,8,9,5,7,2,3}
p3	title	String	10	{3,6,5,4,8,9,5,7,2,3}
p4	genres.1	String	10	{3,6,5,4,8,9,5,7,2,3}
p5	year	Int	10	{3,6,5,4,8,9,5,7,2,3}
p6	awards	Array	10	{3,6,5,4,8,9,5,7,2,3}
p7	ranking	Object	10	{3,6,5,4,8,9,5,7,2,3}
p8	link	String	10	{3,6,5,4,8,9,5,7,2,3}
p9	country	String	10	{3,6,5,4,8,9,5,7,2,3}
p10	score	Float	10	{3,6,5,4,8,9,5,7,2,3}
p11	actors.1	String	10	{3,6,5,4,8,9,5,7,2,3}

- A *Loaded* dataset to evaluate the dictionary construction time on an existing collections. This dataset was composed of five collections of 100 GB containing 2, 4, 6, 8 and 10 schemas respectively;

- An *Adhoc* dataset to evaluate the dictionary construction time for the loading collections phase. This dataset was composed of five collections of 1 GB containing 2, 4, 6, 8 and 10 schemas respectively.

In Table 2 we represent the characteristics of documents in the *Heterogenous* dataset.

In order to have the same results when executing queries across baseline and heterogeneous collections, we carried on using the same values for leaf nodes. The same results imply: (i) the same number of documents, and (ii) the same values for their attributes (leaf nodes). This is why the evaluation did not target result relevance, as the same results will be retrieved by all queries: either homogeneous documents or heterogeneous documents built from homogeneous documents.

5.1.2. Query set

We built two workloads composed of a synthetic series of queries; (i) an *operator evaluation* to evaluate the execution time of reformulated selection-projection-aggregation-unnest-lookup operators, and (ii) an *operator combination evaluation* to evaluate the execution time of the reformulated query composed of operator combination.

The details of the five queries, Q_1, Q_2, Q_3, Q_4, Q_5, from the *operator evaluation* workload are as follows:

- For the projection query we chose to build a query that covers the different options offered for projection operations, e.g., a Boolean expression to compare two paths, project and rename paths, and project paths into an array and the normal projection operation. In addition, we built our query with absolute paths from the baseline collection, e.g., *year, title, director.first_name, lead_actor.first_name* paths for a particular entry in the array, e.g., *genres.1* and leaf

Table 4
The number of extracted documents per the two workloads.

Collection size in GB	# of documents	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8
0.5 GB	1 M	1 M	27 K	66	1 M	1 M	1 M	27 K	23 K
5 GB	10 M	10 M	271 K	66	10.4 M	10 M	10.4 M	271 K	2.3 M
12.5 GB	25 M	25 M	678 K	66	26 M	25 M	26 M	678.7 K	5.7 M
25 GB	50 M	50 M	1.3 M	66	52 M	50 M	52 M	1 M	11.5 M
50 GB	100 M	100 M	2.7 M	66	104 M	100 M	104 M	2.7 M	23 M
250 GB	500 M	500 M	13.5 M	66	521.6 M	500 M	521.6 M	13.5 M	11.5 M

nodes, e.g., *score*. The following is the projection query that we used in our experiments:

$$Q_1 : \pi_{\text{cond:director.first\_name} = \text{lead\_actor.first\_name},}$$
$$\text{desc:[title, genres.1], production\_year:year, score}(C)$$

- For the selection operation we chose to build a query that covers the classical comparison operators, i.e., $\{<, >, \leq, \geq, =, \neq\}$ for numerical values, e.g., (year $\geq$ 2004) as well as classical logical operators, i.e., {and: $\wedge$, or: $\vee$} between query predicates (e.g., ((year $\geq$ 2004) $\vee$ (genres.1 = "Drama"))) Also, we combined these traditional comparisons with a path check condition, e.g., (ranking = {"score": 6 }). The following is the selection query that we used in our experiments:

$$Q_2 : \sigma_{(\text{ year}\geq 2004 \vee \text{genres.1="Drama"})}$$
$$\wedge(\text{ranking={"score ":6}}\vee \text{ link}\neq\text{null })(C)$$

- For the aggregation operation we decided to group movies by *country* and to find the maximum *score* for all movies for each *country*. The following is the aggregation query that we used in our experiments:

$$Q_3 : {}_{\text{country}} \gamma_{\text{maximum\_score:Max(score)}} (C)$$

- We chose to apply the unnest operator to the array *awards* which contains all the awards for a given film. The following is the unnest query that we used in our experiments:

$$Q_4 : \mu_{\text{awards}}(C)$$

- For the lookup operation we decided to generate a new collection, "actors", which is composed of four attributes (*actor*, *birth_year*, *country* and *genre*) with 3033 entries, and we built a lookup query that enriches movie documents with details of the lead actor in each movie. The following is the lookup query that we used in our experiments:

$$Q_5 : (C)\lambda_{\text{ res:actors.1=actor}} (actors)$$

This first workload helped us to separately evaluate each operator. In the second workload *operator combination evaluation* we introduced three additional queries, Q_6, Q_7, Q_8, in which we combined two or more operators. These combinations enabled us to study the effects of operator combinations on the query reformulation and its evaluation by the document query engine. We present these additional queries below:

- We combined the unnest operator from the query "Q_4" with the project operator from query "Q_1":

$$Q_6 : \pi_{\text{ cond:director.first\_name} = \text{lead\_actor.first\_name},}$$
$$\text{desc:[title, genres.1], production\_year:year, score}(\mu_{\text{awards}}(C))$$

Table 5
Number of schema effects on query rewriting (# of paths in reformulated query and reformulation time) and dictionary size (query Q_6) over *Schemas* dataset.

# of schemas	# of absolute paths	Reformulation time	Dictionary size
10	80	0.0005 s	40 KB
100	800	0.0025 s	74 KB
1 k	8 k	0.139 s	2 MB
3 k	24 k	0.6 s	7.2 MB
5 k	40 k	1.52 s	12 MB

- We combined the select operator form query "Q_2" and the project operator from the query "Q_1":

$$Q_7 : \pi_{\text{ cond:director.first\_name} = \text{lead\_actor.first\_name},}$$
$$\text{desc:[title, genres.1], production\_year:year, score)}$$
$$(\sigma_{(\text{year}\geq 2004\vee\text{genres.1=Drama})}$$
$$\wedge(\text{ranking }={\text{score:6}}\vee(\text{link }\neq\text{null})(C))$$

- We combined the select operator from query "Q_2", the unnest operator from query "Q_4" and the project operator from query "Q_1":

$$Q_8 : \pi_{\text{cond:director.first\_name} = \text{lead\_actor.first\_name},}$$
$$\text{desc:[title, genres.1], production\_year:year, score}$$
$$(\sigma_{(\text{ year}\geq 2004\vee\text{genres. 1=Drama})}$$
$$\wedge(\text{ranking={"score":6}}\vee \text{ link}\neq\text{null})(\mu_{\text{awards}}(C)))$$

Table 3 highlights the different characteristics of the selected attributes in queries from both workloads and gives details about their depth inside the structurally heterogeneous collection.

In our queries we employed 11 attributes of different types (primitive and complex) and different depths ranging from 2 to 9 intermediary attributes that should be traversed to reach the attributes containing data of interest. Also, we represented the paths in several ways, e.g., absolute paths, array entries, relative paths and leaf node. Table 4 gives the number of documents to be retrieved for each query.

The query reformulation process replaces each element with its 10 corresponding paths. For instance, the query $Q_{1,ext}$ contains 60 absolute paths for its 6 initial paths, 10 in query $Q_{4,ext}$ etc.

We describe three contexts for which we ran the queries as defined above. For the purpose of this experiment we used the *Baseline* dataset to study the classical query engine execution time for both workloads. Furthermore, we used the *Heterogeneous* dataset to evaluate the execution time of reformulated queries from both workloads. For each context we measured the average execution duration after executing each query at least five times. The query execution order was random.

We present the details of the three evaluation contexts as follows:

- Q_{Base} is the name of the query that refers to the initial user query (one of the queries from above): it was executed across the *Baseline* dataset. The purpose of this first context was to study the native behaviour of the document store. We used this first context as a baseline for our experimentation;

- Q_{Ext} refers to the query Q_{Base} reformulated by our approach. It was executed across the *Heterogeneous* dataset;
- $Q_{Accumulated}$ refers to distinct queries where each query is formulated for a single schema found in the collection. In our case we needed 10 separated queries as we were dealing with collections with ten schemas. These queries were built manually without any additional tools. We did not consider the time required to merge the results of each query as we were more interested in measuring the time required to retrieve relevant documents. We executed each of the $Q_{Accumulated}$ across the *Heterogeneous* dataset. The result was therefore the accumulated time required to process the 10 queries.

5.1.3. Queries evaluation results

As shown in Fig. 5, we can see that our reformulated query, Q_{Ext}, outperforms the accumulated query, $Q_{Accumulated}$, for all queries. The difference between the two execution scenarios comes from the ability of our query to automatically include all corresponding absolute paths for the different query elements. Hence, the query is executed only once when the accumulated query requires several passes through the collection (10 passes). This solution requires more CPU loads and more intensive disk I/O operations. We now examine the efficiency of the reformulated query when compared to the baseline query Q_{Base}. We can see that the overhead of our solution is up to three times more, e.g., projection, selection and unnest when compared to the native execution of the baseline query on the *Baseline* dataset. Moreover, we score an overhead that does not exceed a multiple of two in the evaluation of the aggregation operator. We believe that this overhead is acceptable as we can bypass the costs needed for refactoring the underlying data structures, similarly to other state-of-the-art research work. Unlike the baseline, our *Heterogeneous* dataset contains different grouping objects with varying nesting levels. Therefore, the rewritten query includes several navigational paths which were processed by the native query engine, MongoDB, to find matches for each visited document among the collection. Finally, we must emphasize that the execution time for the lookup operators is very similar between Q_{Base} and our reformulated query Q_{Ext}.

We do not present the $Q_{Accumulated}$ evaluation for the query Q_5 from the *operator evaluation* workload and the *operator combination evaluation* workload due to the complexity and the considerable number of accessed collections required to evaluate the $Q_{Accumulated}$ context. For example, to evaluate the query Q_8 from the second workload, we would need to build 30 separate queries. Therefore, we would need to go through the collection 30 times. Furthermore, it is complicated to combine the results. Thus, this process is difficult and time-consuming, and combining partial results may lead to corrupted results.

In Fig. 6, we compare the time required to execute Q_{Ext} with the time required to execute Q_{Base} when the query is a combination of operators. It is notable that the overhead arising from the evaluation of our reformulated query is the same as the overhead arising from the execution of the standalone operator (around three times the size when compared to querying a heterogeneous collection).

This series of workload evaluations shows that the overhead for the time required to evaluate our reformulated query is linear with the increasing number of documents. The same behaviour for the native query occurs when studying the effects of the scalability on the query evaluation. Furthermore, the overhead induced by the evaluation of our reformulated query is not affected by the number of documents or the combination of operators.

Furthermore, we executed the query Q_6 from the *operator combination evaluation* workload over the *Structures* dataset: we

Table 6
Evaluating Q_6 on varying number of schemas, *Structures* dataset.

# of Schemas	10	20	50	100	200
Time in (s)	200	380	690	1140	2560

Table 7
Time to build the dictionary for collections from the *Loaded* dataset.

# of schema	2	4	6	8	10
Required time (min)	96	108	127	143	156
Size of the resulting dictionary (KB)	4154	9458	13,587	17,478	22,997

present the time needed to execute the reformulated query in Table 6. This experiment helps us to study the effect of executing our reformulated query on the varying number of schemas. It is notable that the time evolves linearly rather than exponentially as more heterogeneity is added. This is due also to the important number of comparison required waiting for the one query. For instance, the execution of the query Q_6 over the collection having 200 schemas requires 200 possible paths for each attribute.

5.2. Query reformulation evaluation

For this experiment we only executed the query Q_6 from the *operator combination evaluation* workload over the *Schemas* dataset: we present the time needed to build the reformulated query in Table 5. It is notable that the time to generate the reformulated query is less than two seconds, which is very low. Also, it is possible to construct a dictionary that covers a collection of heterogeneous documents. Here our dictionary can support up to 5000 distinct schemas, which is the limit for the number of schemas we decided on for the purpose of this experiment. The resulting size of the materialized dictionary is very promising because it does not require significant storage space. Furthermore, we also believe that the time required to reformulate the query is of great interest and represents another advantage of our solution. In this series of experiments we have tried to find distinct navigational paths for eight predicates. Each rewritten query is composed of numerous disjunctive forms for each predicate. Table 5 shows 80 disjunctive forms for datasets containing 10 schemas, 800 disjunctive forms for 100 schemas, 8000 for 1000 schemas, 24,000 for 3000 schemas and 40,000 for 5000 schemas. We believe that the dictionary and the query rewriting engine scale effectively when handling heterogeneous collections which contain a high number of schemas. We succeeded in executing all the reformulated queries on MongoDB. We noticed a limitation in terms of performance: the execution time can be 50 times more than the execution of similar queries on the *Baseline* dataset. This limitation is due to the considerable number of comparisons per document. In the worst case scenario we would need to perform 40,000 comparisons per document when dealing with a collection containing 5000 distinct schemas.

5.3. Dictionary construction time

In this part we focus on the dictionary construction process. *EasyQ* offers the possibility of building the dictionary from existing collections or during the data loading phase. We studied both configurations. First we evaluated the time required to build the dictionary for collections from the *Loaded* dataset.

We can see from the results in Table 7 that the time taken to build the dictionary increases when we start to deal with collections which have more heterogeneity. When a collection has 10 structures, the time does not exceed 40% when we compare it to a collection with 2 structures. In Table 7 we can again see
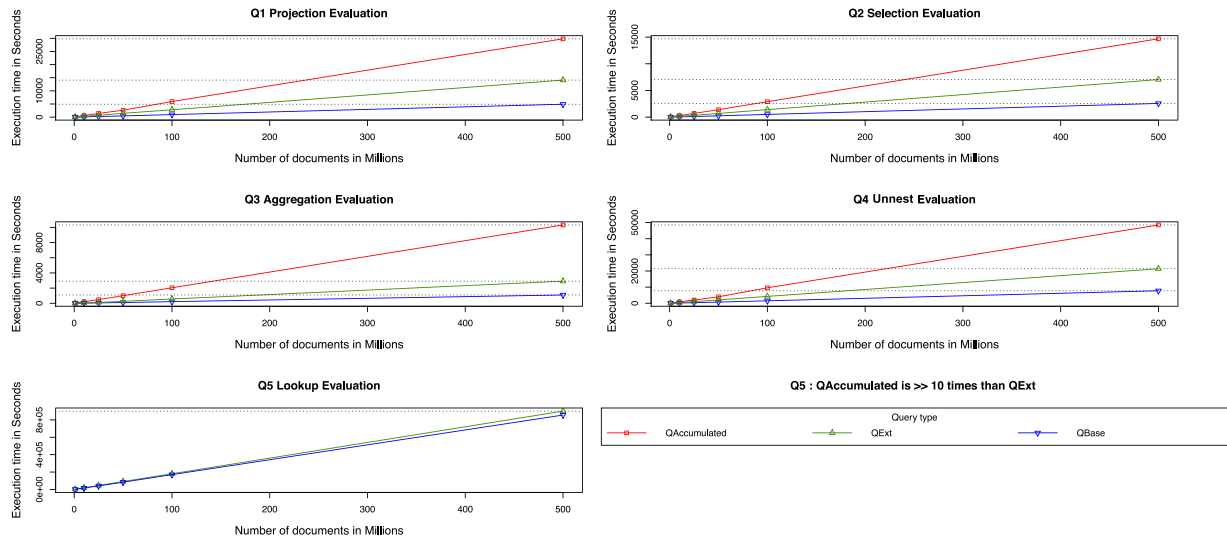
Fig. 5. Query reformulation evaluation.
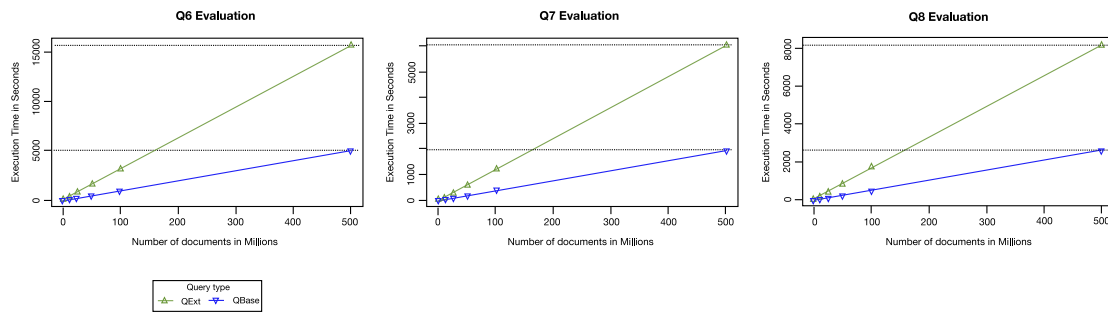
Fig. 6. Query combination evaluation.

Table 8
Study of the overhead added during load time.

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

the negligible size of the generated dictionaries when compared to the 100 GB of the collection. Afterwards we wanted to evaluate the overhead that causes the generation of the dictionary at loading time using the *Adhoc* dataset. We present two measurements in Table 8. First, we measured the time to classically load each collection (without the dictionary building). Second, we measured the overall time to build the dictionary while loading the collection.

In this experiment we found that the overhead measure does not exceed 50% of the time required to only load data. The time evolves linearly rather than exponentially as more heterogeneity is added, which is encouraging. Many factors may affect the dictionary construction phase. The number of attributes and the nesting levels may increase or decrease the overhead. The advantage of our solution is that once the data is loaded and the dictionary is built or updated, the rewritten query automatically takes all the changes into account.

6. Conclusion

NoSQL document stores are often called schemaless because they may contain variable schemas among stored data. Nowadays, this variability is becoming a common feature of many applications, such as web applications, social media applications and the internet of things. Nevertheless, the existence of structural heterogeneity makes it very hard for users to formulate queries that achieve relevant and coherent results.

In this paper we have presented *EasyQ*, an automatic mechanism which enables schema-independent querying for multi-structured document stores. To the best of our knowledge, *EasyQ* is the first mechanism of its kind to offer schema-independent querying without the need to learn new querying languages and new structures, or to perform heavy transformation on the underlying document structures.

Our contribution is based on a constructed dictionary which matches each possible partial path, leaf node and absolute path with its corresponding absolute paths among the different document structures inside the collection. Using this dictionary, we can apply reformulation rules to rewrite the user query and find relevant results. The query reformulation can be applied to most document store operators based on formal foundations that are stated in the paper.

In our experiments we compared the execution time cost of basic MongoDB queries and rewritten queries proposed by our approach. We conducted a set of tests by changing the size of the dataset and the structural heterogeneity inside a collection (number of grouping levels and nesting levels). Results show that the cost of executing the rewritten queries proposed in this

paper is higher when compared to the execution of basic user queries, but always less than a multiple of three. Nevertheless, this time overhead is acceptable when compared to the execution of separated (manually built) queries for each schema while heterogeneity issues are automatically managed.

Our approach is a syntactic manipulation of queries, so it is based on an important assumption: the collection describes *homogeneous entities*, i.e., a field may have the same meaning in all document schemas. In case of ambiguity, the user should specify a sub-path (partial path) in order to overcome this ambiguity. If this assumption is not guaranteed, users may obtain irrelevant results. Nevertheless, this assumption may be acceptable in many applications, such as legacy collections, web applications and IoT data.

One novel aspect of our proposal is that we have provided a generic reformulation mechanism based on a dictionary. For the scope of this paper, the dictionary is built and updated automatically. Nevertheless, the dictionary content may be defined specifically for a given application in order to target specific heterogeneity. The reformulation mechanism remains generic for all applications whereas dictionaries can be tailored to specific needs.

This paper contrasts with classical documents stores in that we offer users the ability to query documents using partial paths and thus EasyQ manages to find all information regardless of the document structures. Furthermore, by using specific dictionaries we extend the native querying capabilities of document stores, even when querying homogeneous documents.

Another original aspect is that any query will always return relevant and complete data whatever the state of the collection. Indeed, the query is reformulated each time it is evaluated. If new heterogeneous documents have been added to the collection, their schemas are integrated into the dictionary and the reformulated query will cover these new structures too.

Future research work will cover the different aspects presented in this paper. Initial research will focus on testing *EasyQ* on more complex queries and ever larger datasets. We also plan to employ our mechanism on real data-intensive applications. For the query reformulation process we will enable support for more document operations. This will cover all "C.R.U.D." operations such as "update" and "delete" to overcome structural heterogeneity in document stores. Moreover, we will work on the interaction between the user and our systems so that the user has the possibility of selecting certain absolute paths or removing unnecessary absolute paths, e.g., because a multi-entity has collapsed in the reformulated query, which will assist our mechanism while reformulating the initial user query. A long-term aim will be to cover most classes of heterogeneity, e.g., syntactic and semantic classes, and thus provide different dictionary building processes.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] R. Hecht, S. Jablonski, Nosql evaluation: A use case oriented survey, in: Cloud and Service Computing, CSC, 2011 International Conference on, IEEE, 2011, pp. 336–341.

[2] A. Floratou, N. Teletia, D.J. DeWitt, J.M. Patel, D. Zhang, Can the elephants handle the nosql onslaught? Proc. VLDB Endow. 5 (12) (2012) 1712–1723.

[3] M. Stonebraker, New opportunities for new sql, Commun. ACM 5 (11) (2012) 10–11.

[4] P. Shvaiko, J. Euzenat, A survey of schema-based matching approaches, J. Data Semant. IV (2005) 146–171.

[5] E. Rahm, P.A. Bernstein, A survey of approaches to automatic schema matching, VLDB J. 10 (4) (2001) 334–350.

[6] D. Tahara, T. Diamond, D.J. Abadi, Sinew: a sql system for multi-structured data, in: Proceedings of the 2014 ACM SIGMOD, ACM, 2014, pp. 815–826.

[7] L. Wang, S. Zhang, J. Shi, L. Jiao, Hassanzadeh, Schema management for document stores, Proc. VLDB Endow. 8 (9) (2015) 922–933.

[8] D. Florescu, G. Fourny, Jsoniq: The history of a query language, IEEE Internet Comput. 17 (5) (2013) 86–90.

[9] K. Chodorow, MongoDB: The Definitive Guide: Powerful and Scalable Data Storage, O'Reilly Media, Inc., 2013.

[10] J.C. Anderson, J. Lehnardt, N. Slater, CouchDB: The Definitive Guide: Time to Relax, O'Reilly Media, Inc., 2010.

[11] J. Murty, Programming Amazon Web Services: S3, EC2, SQS, FPS, and SimpleDB, O'Reilly Media, Inc., 2008.

[12] C. Chasseur, Y. Li, J.M. Patel, Enabling json document stores in relational systems, in: WebDB, vol. 13, 2013, pp. 14–15.

[13] M. DiScala, D.J. Abadi, Automatic generation of normalized relational schemas from nested key–value data, in: Proceedings of the 2016 International Conference on Management of Data, ACM, 2016, pp. 295–310.

[14] R. Hai, S. Geisler, C. Quix, Constance: An intelligent data lake system, in: Proceedings of the 2016 International Conference on Management of Data, ACM, 2016, pp. 2097–2100.

[15] M.-A. Baazizi, H.B. Lahmar, D. Colazzo, G. Ghelli, C. Sartiani, Schema inference for massive json datasets, in: EDBT, 2017, pp. 222–233.

[16] D.S. Ruiz, S.F. Morales, J.G. Molina, Inferring versioned schemas from nosql databases and its applications, in: International Conference on Conceptual Modeling, Springer, 2015, pp. 467–480.

[17] K.W. Ong, Y. Papakonstantinou, R. Vernoux, The sql++ query language: Configurable, unifying and semi-structured, arXiv preprint arXiv:1405. 3631.

[18] S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, J. Siméon, M. Stefanescu, Xquery 1.0: An xml query language.

[19] A.P. Sheth, J.A. Larson, Federated database systems for managing distributed, heterogeneous, and autonomous databases, ACM Comput. Surv. 22 (3) (1990) 183–236.

[20] A. Corbellini, C. Mateos, A. Zunino, D. Godoy, S. Schiaffino, Persisting big-data: The nosql landscape, Inf. Syst. 63 (2017) 1–23.

[21] H. Ben Hamadou, F. Ghozzi, A. Péninou, O. Teste, Towards schema-independent querying on document data stores, in: Proceedings of the 20th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP), Vienna, Austria, March 26–29, 2018. 2018.

[22] H.-H. Do, E. Rahm, Coma—a system for flexible combination of schema matching approaches, in: VLDB'02: Proceedings of the 28th International Conference on Very Large Databases, Elsevier, 2002, pp. 610–621.

[23] J. Wang, J.-R. Wen, F. Lochovsky, W.-Y. Ma, Instance-based schema matching for web databases by domain-specific query probing, in: Proceedings of the Thirtieth International Conference on Very Large Data Bases-Volume 30, VLDB Endowment, 2004, pp. 408–419.

[24] P.A. Hall, G.R. Dowling, Approximate string matching, ACM Comput. Surv. 12 (4) (1980) 381–402.

[25] E.M. Voorhees, Using wordnet to disambiguate word senses for text retrieval, in: Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM, 1993, pp. 171–180.

[26] V. Herrero, A. Abelló, O. Romero, Nosql design for analytical workloads: variability matters, in: ER 2016, Gifu, Japan, November 14–17, 2016, Proceedings 35, Springer, 2016, pp. 50–64.

[27] E. Gallinucci, M. Golfarelli, S. Rizzi, Schema profiling of document-oriented databases, Inf. Syst. 75 (2018) 13–25.

[28] Y. Papakonstantinou, V. Vassalos, Query rewriting for semistructured data, in: ACM SIGMOD Record, vol. 28, ACM, 1999, pp. 455–466.

[29] C. Lin, J. Wang, C. Rong, Towards heterogeneous keyword search, in: Proceedings of the ACM Turing 50th Celebration Conference-China, ACM, 2017, p. 46.

[30] J. Clark, S. DeRose, et al., XML path language (xpath) version 1.0, 1999.

[31] H. Ben Hamadou, F. Ghozzi, A. Péninou, O. Teste, Querying heterogeneous document stores, in: Proceedings of the 20th International Conference on Enterprise Information Systems, ICEIS 2018, Funchal, Madeira, Portugal, March 21–24, 2018, vol. 1, 2018, pp. 58–68.

[32] P. Bourhis, J.L. Reutter, F. Suárez, D. Vrgoč, Json:.data. model, Json: data model, query languages and schema specification, in: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, ACM, 2017, pp. 123–135.

[33] J. Hidders, J. Paredaens, J. Van den Bussche, J-logic: Logical foundations for json querying, in: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, ACM, 2017, pp. 137–149.

[34] E. Botoeva, D. Calvanese, B. Cogrel, G. Xiao, Expressivity and complexity of mongodb queries, in: 21st International Conference on Database Theory, ICDT 2018, March 26–29, 2018, Vienna, Austria, 2018, pp. 9:1–9:23.