Querying heterogeneous data in NoSQL document stores

Hamdi Ben Hamadou

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *02/10/2019* **par :**

Hamdi Ben Hamadou

Querying Heterogeneous Data in NoSQL Document Stores

JURY

Bernd AMANN	Professeur, LIP6-Université Sorbonne	Examinateur
Omar BOUSSAID	Professeur, Université Lyon 2	Rapporteur
Faiza GHOZZI	Maître Assistant, Université de Sfax	Examinatrice
Anne LAURENT	Professeur, LIRMM-Montpelier	Rapporteure
André PÉNINOU	MC, Université Toulouse UT2J	Co-directeur
Franck RAVAT	Professeur, Université Toulouse UT1C	Examinateur
Olivier TESTE	Professeur, Université Toulouse UT2J	Directeur
Isabelle Comyn-Wattiau	Professeure, ESSEC Business School	Examinatrice

---

École doctorale et spécialité :
MITT : Domaine STIC : Sureté de logiciel et calcul de haute performance

Unité de Recherche :
Institut de Recherche en Informatique de Toulouse (UMR 5505)

Directeur(s) de Thèse :
Olivier TESTE et *André* PÉNINOU

Rapporteurs :
Omar BOUSSAID et *Anne* LAURENT

Acknowledgement

Firstly, I would like to thank my family: my parents and my sister for supporting me spiritually throughout all the years of my studies, for accepting the fact that I am abroad past three years and for all their love and encouragement. For my parents who raised me with a love of science and supported me in all my orientations.

I would like to express my sincere gratitude to my advisor Prof. Olivier TESTE for the continuous support of my PhD study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor for my PhD study.

Besides my advisor, I would like to thank my thesis co-advisors: Dr Faiza GHOZZI, and Dr André Péninou, for their insightful comments and encouragement, but also for the hard question which motivated me to widen my research from various perspectives.

My sincere thanks also go the reviewers, Pr. Omar BOUSSAID, and Pr. Anne LAURENT, to whom I associate the members Pr. Bernd AMANN, Pr. Franck RAVAT, and Pr. Isabelle Comyn-Wattiau for taking time to evaluate my work.

I gratefully acknowledge the funding received towards my PhD from the neOCampus project PhD fellowship. Thanks to Pr. Marie-Pierre Gleizes for her encouragement.

I thank my colleagues at IRIT for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last three years.

Last but not least, I would like to thank the one that she believed in me. I would like to thank you for being part of my achievements, thank you for assisting me, standing with me and encouraging me to go further.
I would like to say thank you to all who is dear to me.

Resumé

La problématique de cette thèse porte sur l'interrogation de données hétérogènes dans les systèmes de stockage « not-only SQL » (noSQL) orientés documents. Ces derniers ont connu un important développement ces dernières années en raison de leur capacité à gérer de manière flexible et efficace d'importantes masses de documents. Ils reposent sur le principe « schema-less » consistant à ne plus considérer un schéma unique pour un ensemble de données, appelé collection de documents. Cette flexibilité dans la structuration des données complexifie l'interrogation pour les utilisateurs qui doivent connaître l'ensemble des différents schémas des données manipulées lors de l'écriture de requêtes.

Les travaux développés dans cette thèse sont menés dans le cadre du projet neoCampus. Ils se focalisent sur l'interrogation de documents structurellement hétérogènes, en particulier sur le problème de schémas variables. Nous proposons la construction d'un dictionnaire de données qui permet de retrouver tous les schémas des documents. Chaque clef, entrée du dictionnaire, correspond à un chemin absolu ou partiel existant dans au moins un document de la collection. Cette clef est associée aux différents chemins absolus correspondants dans l'ensemble de la collection de documents. Le dictionnaire est alors exploité pour réécrire de manière automatique et transparente les requêtes des utilisateurs. Les requêtes utilisateurs sont établies sur la base des clés du dictionnaire (chemins partiels ou absolus) et sont automatiquement réécrites en exploitant le dictionnaire afin de prendre en compte l'ensemble des chemins absolus existants dans les documents de la collection.

Dans cette thèse, nous menons une étude de l'état de l'art des travaux s'attachant à résoudre l'interrogation de documents structurellement hétérogènes, et nous en proposons une classification. Ensuite, nous comparons ces travaux en fonction de critères qui permettent de positionner et différencier notre contribution. Nous définissions formellement les concepts classiques liés aux systèmes orientés documents (document, collection, etc), puis nous étendons cette formalisation par des concepts supplémentaires : chemins absolus et partiels, schémas de document, dictionnaire. Pour la manipulation et l'interrogation des documents, nous définissons un noyau algébrique minimal fermé composé de cinq opérateurs : sélection, projection, des-imbrication (unnest), agrégation et jointure (left-join). Nous définissons chaque opérateur et expliquons son évaluation par un moteur de requête classique. Ensuite, nous établissons la réécriture de chacun des opérateurs à partir du dictionnaire. Nous définissons le processus de réécriture des requêtes utilisateurs qui produit une requête évaluable par un moteur de requête classique en conservant la logique des opérateurs classiques (chemins inexistants, valeurs nulles). Nous montrons comment la réécriture d'une requête initialement construite avec des chemins partiels et/ou absolus permet de résoudre le problème d'hétérogénéité structurelle des documents.

Enfin, nous menons des expérimentations afin de valider les concepts formels que nous introduisons tout au long de cette thèse. Nous évaluons la construction et la maintenance du dictionnaire en changeant la configuration en termes de nombre de structures par collection étudiée et de taille de collection. Puis, nous évaluons le moteur de réécriture de requêtes en le comparant à une évaluation de requête dans un contexte sans hétérogénéité structurelle puis dans un contexte de multi-requêtes. Toutes nos expérimentations ont été menées sur des collection synthétiques avec plusieurs niveaux d'imbrications, différents nombres de structure par collection, et différentes tailles de collections. Récemment, nous avons intégré notre contribution dans le projet neO-Campus afin de gérer l'hétérogénéité lors de l'interrogation des données de capteurs implantés dans le campus de l'université Toulouse III-Paul Sabatier.

Summary

This thesis discusses the problems related to querying heterogeneous data in document-oriented systems. Document-oriented "not-only SQL" (noSQL) storage systems have undergone significant development in recent years due to their ability to manage large amounts of documents in a flexible and efficient manner. These systems rely on the “schema-less" concept where no there is no requirement to consider a single schema for a set of data, called a collection of documents. This flexibility in data structures makes the query formulation more complex and users need to know all the different schemas of the data manipulated during the query formulation.

The work developed in this thesis subscribes into the frame of neOCampus project. It focuses on issues in the manipulation and the querying of structurally heterogeneous document collections, mainly the problem of variable schemas. We propose the construction of a dictionary of data that makes it possible to find all the schemas of the documents. Each key, a dictionary entry, corresponds to an absolute or partial path existing in at least one document of the collection. This key is associated to all the corresponding absolute paths throughout the collection of heterogeneous documents. The dictionary is then exploited to automatically and transparently reformulate queries from users. The user queries are formulated using the dictionary keys (partial or absolute paths) and are automatically reformulated using the dictionary to consider all the existing paths in all documents in the collection.

In this thesis, we conduct a state-of-the-art survey of the work related to solving the problem of querying data of heterogeneous structures, and we propose a classification. Then, we compare these works according to criteria that make it possible to position our contribution. We formally define the classical concepts related to document-oriented systems (document, collection, etc). Then, we extend this formalisation with additional concepts: absolute and partial paths, document schemas, dictionary. For manipulating and querying heterogeneous documents, we define a closed minimal algebraic kernel composed of five operators: selection, projection, unnest, aggregation and join (left-join). We define each operator and explain its classical evaluation by the native document querying engine. Then we establish the reformulation rules of each of these operators based on the use of the dictionary. We define the process of reformulating user queries that produces a query that can be evaluated by most document querying engines while keeping the logic of the classical operators (misleading paths, null values). We show how the reformulation of a query initially constructed with partial and / or absolute paths makes it possible to solve the problem of structural heterogeneity of documents.

Finally, we conduct experiments to validate the formal concepts that we introduce throughout this thesis. We evaluate the construction and maintenance of the dictionary by changing the configuration in terms of number of structures per collection

studied and collection size. Then, we evaluate the query reformulation engine by comparing it to a query evaluation in a context without structural heterogeneity and then in a context of executing multiple queries. All our experiments were conducted on synthetic collections with several levels of nesting, different numbers of structures per collection, and on varying collection sizes. Recently, we deployed our contributions in the neOCampus project to query heterogeneous sensors data installed at different classrooms and the library at the campus of the university of Toulouse III-Paul Sabatier.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Research Context

Big Data has emerged as evolving term in both the academic and the business communities over the past two decades. Under the explosive increase of generated data, the term of big data is mainly used to describe large and complex data caracterised by three aspects: *i)* variety: data comes in all types of formats, e.g., structured, semi-structured and unstructured, *ii)* velocity: the speed with which data are being generated to be ingested and *iii)* volume: the enormous amount of generated data. These aspects were introduced by Gartner [1] to describe the big data challenges.

Such voluminous data can come from myriad different sources, such as business transaction systems, customer databases, medical records, internet clickstream logs, mobile applications, social networks, the collected results of scientific experiments, machine-generated data and real-time data sensors used in the internet of things (IoT) environments (Chen and Zhang, 2014). Data may be left in its raw form or pre-processed using data mining tools or data preparation software before being analysed. Thus, data arriving from different sources and representing the same kind of information do not necessarily share the same structure. Furthermore, data structures are not stable and are subjects to future changes. Such structure evolution appears as applications evolve and change for many reasons: systems evolution, systems maintenance, diversity of data sources, data enrichment over time, etc.

NoSQL, which stand for "not only SQL", databases and schema-less data modelling have emerged as mainstream database alternatives for addressing the substantive requirements of current data-intensive applications (Hecht and Jablonski, 2011). NoSQL is an approach to database design that can essentially accommodate four data models, including key-value, document, column and graph model. NoSQL, is an alternative to conventional relational databases in which data is placed in tables and data schema should be carefully designed before building the database. NoSQL databases are especially useful for working with large volume of data in distributed and fault-tolerant environments (Chevalier et al., 2015).

Schema heterogeneity is common feature in most NoSQL systems as they abandon the traditional "schema first, data later" approach of RDBMS, which requires all record in a table to comply to a certain predefined fixed schema, in favour of a "schema-less" approach. NoSQL schema-less database can store data with different structure for the same entity type. Furthermore, they do not require to define a rigid schema, e.g., database, schema, data types, or tables. The lack of data structure restrictions enables easy evolution of data and facilitates the integration of data from different sources.

Document stores are one of the main NoSQL data models designed to store, retrieve and manage collections of documents of JSON objects. JSON (JavaScript Object Notation) is a format for formatting semi-structured data in human-readable text. The

[1] https://www.gartner.com/

usage of JSON objects in the document data model offers several advantages (Chasseur et al., 2013): *i)* Ease-of-use since it is not required to define a schema upfront, *ii)* Sparseness where attributes could appear in some documents, but not in others, *iii)* Hierarchical data where information could be present at different nesting level within the document, and *iv)* Dynamic typing where types of the values of attributes can be different for each record. Furthermore, schema-less document stores are able to store documents in transparent and efficient ways (Floratou et al., 2012, Stonebraker, 2012).

Despite the flexibility guaranteed by most of the schema-less NoSQL stores while loading data, querying multi-structured collection in document stores is a burdensome task. In document stores, formulating relevant queries require full knowledge of the underlying document schemas. Generally, queries are formulated only over absolute paths, i.e., paths starting from the root of the document to the attribute of interest. Most document stores adopt this assumption (e.g., MongoDB, CouchDB, Terrastore (Anderson et al., 2010, Chodorow, 2013, Murty, 2008)). Whereas, the presence of several schemas within the same collection requires to explicitly use in the query as many paths as the variation for the same attribute. This condition makes it difficult to define adequate workloads.

In this thesis we address this problematic and we propose solutions to facilitate the querying of heterogeneous collection of documents while omitting the limitation in state-of-the-art solutions.

1.1.1 Running Example

Figure 1.2 illustrates a collection composed of four documents (a, b, c, d) in JSON format. 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 can be seen as 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. Figure 1.1 illustrates the hierarchical representation of document (a).

1.1.2 Heterogeneity Classes

Several kinds of heterogeneity are discussed in the literature (Rahm and Bernstein, 2001): structural heterogeneity refers to diverse possible locations of attributes within a collection due to documents diverse structures, e.g., nested or flat structures and different nesting levels as shown in Figure 1.2; syntactic heterogeneity refers to dif-

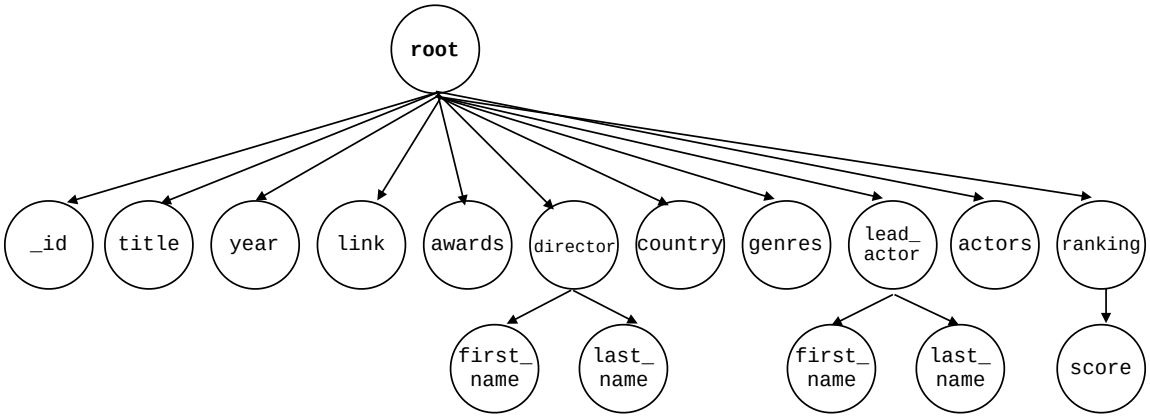

Figure 1.1: Hierarchical representation of the document (a)

ferences in the representation of the attributes, e.g., usage of acronyms, prefix, suffix and special characters due to several naming convention affecting specifically attribute names, e.g., *movie_title* or *movieTitle*; finally, semantic heterogeneity may exist when the same field relies on distinct concepts in separate documents, e.g., *country* and *nation* (Shvaiko and Euzenat, 2005).

1.1.3 The Problem of Structural Heterogeneity

For the aim of this work, we build our assumptions to resolve the problem of structural heterogeneity to enable querying multi-structured collections. Several works were conducted to resolve the semantic and syntactic heterogeneities (Rahm and Bernstein, 2001). The structural heterogeneity refers to the presence of several paths allowing to retrieve information of a given attribute. Because of their flexibility, an attribute within a collection could be root based, nested within another attribute or array of attributes. Furthermore, it is possible that an attribute is nested at different depth, several nesting levels. Thus, the lack of knowledge of full locations corresponding to each attribute within a collection increase the difficulty to fetch relevant results. The Figure 1.2 highlights this problem. 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. Furthermore, *description.title* represents a fourth absolute path in document (d) for the *title* information.

To retrieve information from a document attribute from a document stores, it is necessary to build queries using the absolute path composed of all attribute to cross starting from the document root down to the attribute of interest. If a user formulates

```
{ "_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)

Figure 1.2: Illustrative example of a collection (C) with four documents describing films.

a projection query using only the absolute path *title*, any document query 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 all information related to the attribute *year*, the user should know about the distinct absolute paths as 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 the following 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 notice that information related to *ranking.score* for document (a) is only present on 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, to query multi-structured data, and using several queries to target different paths, the user has to consider making an effort to merge results, to learn the underlying data structures, and to remove possibly redundant information in queries results. Another way is to combine all possible paths in a single query. In this example, a possible query to consider all 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 only two pieces of information, i.e., *title* and *ranking.score.*

This problem is not only limited for project operator as mention in this section. However, other operators could be affected by structural heterogeneity, e.g., select, also called restrict, operator. Hence, if the user formulates a query to retrieve movies having release year greater than the year *2000* using the path *year* from document (a), the result contains only document (a) whereas the document (c) satisfies the selection condition. The problem is that the information related to the *year* is reachable using another path *details.year*. Therefore, the query should include both paths. A possible single query could be (*year = 2000* or *info.year = 2000* or *film.details.year = 2000* or *description.year = 2000*) which is yet a complex query to satisfy a simple user need.

1.2 Research Problems

During the last decade, NoSQL databases and schema-less data modelling have emerged as mainstream alternatives to relational modelling for addressing the substantive requirements of current data-intensive applications (Hecht and Jablonski, 2011), 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 (Floratou et al., 2012, Stonebraker, 2012). Nevertheless, it is possible to store a set of heterogeneous documents inside the same collection, and for the purposes of this thesis, 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.

This thesis aims at answering the following research question: "How to enable schema-independent querying for heterogeneous documents in NoSQL document stores?

With respect to state-of-the-art, there are two issues involved: *i)* Schema transformations, e.g., physical, to resolve schemas variety within a collection of documents, and *ii)* query expressiveness on varying schemas of documents. Due to the complexity involving both subjects, we start our research by exploring limitations regarding two research problems: *1)* Schema transformation; *2)* Query expressiveness.

Problem 1: Schema Transformation

The basic idea of transforming schemas in multi-structured collections is to overcome the heterogeneity in documents structures within a collection. With each manipulation operation, i.e., insert, update and delete, the schema of documents may be changed. The challenge arises when there is a need to have access to information in such heterogeneous documents. Thus, documents querying engine requires that queries are explicitly formulated over all existing absolute paths leading to the information of interest. However, current techniques recommend flattening documents, e.g., in XML or JSON format, into a relational form (Chasseur et al., 2013, DiScala and Abadi, 2016, Tahara et al., 2014) to overcome the heterogeneity in structures. This process of physical schema transformation requires additional resources, such as an external relational database and more effort to generate new schema every time they change the workload or when new data are inserted, deleted or updated. Furthermore, to deal with the heterogeneity in structures they propose transforming all document schemas to a single common schema and introducing some logical views which leads to a homogeneous collection (Tahara et al., 2014) using schema matching techniques to merge

heterogeneous structures (Rahm and Bernstein, 2001). These work are detailed in Chapter 2.

Problem 2: Query Expressiveness

The problem of query expressiveness is the power of querying language to overcome the underlying heterogeneity in documents. Work has been conducted to ensure that document data model could be queried without any prior schema validation or restriction and requires the user to take heterogeneity into account (Wang et al., 2015). However, several directions exist in the literature to provide uniform data access thus to enable querying such heterogeneous documents. First line of work relies on the native query expressiveness of the underlying document stores. Therefore, to formulate queries based on relational views built on top of the inferred data structures whereas these queries should change when new data are inserted (or updated) in the collection. Other work defines a new querying mechanism using complex syntax to make transparent the structural heterogeneity in the data (Florescu and Fourny, 2013). Details regarding these work are introduced in Chapter 2.

Given these considerations, we propose a novel approach based on query reformulation to enable schema-independent querying for heterogeneous collection of documents without the need to change the original document structures neither the application workloads. Therefore, to enable our novel approach, we target to resolve two main problems; *i)* extracting paths from multi-structured collections, and *ii)* enabling schema-independent querying for multi-structured collections.

1.3 Thesis Contributions

In order to enable schema-independent querying for heterogeneous documents in NoSQL document stores, we start first by drawing some assumptions:

- *no physical document transformation*: We advocate the idea of keeping the documents on their original schemas and under the original physical storage model. Thus, documents are queried regardless of their structures.
- *not only absolute paths in queries*: Using only absolute paths while formulating queries presents some limitations while dealing with collection of documents having structural heterogeneity. User queries are expressed according to the locations of the attributes inside the documents. We advocate that user queries should be expressed according to user needs considering data content and not locations. Thus, queries should be expressed using leaf attributes leading to

content, e.g., *year* in Figure 1.2 or partial paths in order to avoid leaf attributes ambiguity. For instance, in Figure 1.2 the path *last_name* refers to *director* or *lead_actor*. A partial path such as *director.last_name* allow to solve this ambiguity and target the content corresponding to the user needs.

- *using native document store query engine:* Document stores offer efficient query engines. Processing document using third party programs outside the document stores would be a time and space consuming tasks requiring dealing with all documents within a collection. Thus, performances of such execution context would be not optimised when compared to processing the data directly into their underlying stores. Therefore, we advocate the idea to build queries that could be executed in most document stores using their underlying query engines.

Considering these three assumptions, we propose the following process for schema-independent querying:

1. users express their needs using queries formulated over partial paths to retrieve requested information.

2. the user query must be extended to include all possible absolute paths existing within the collection of heterogeneous documents. Therefore, the user query is automatically reformulated to replace each partial path with its corresponding absolute paths in the different structures. We refer to the latter query as extended query.

   Thus, we introduce a dictionary that enables to map any partial path that may be found in diverse document structures to their corresponding absolute paths as found in the different structures.

3. finally, the extended query is executed using the native query engine of the underlying document store.

This PhD research resulted in two main contributions summarised as follows:

- meanwhile current trends suggest performing complex physical transformation, we advocate the idea of keeping on the documents in their original underlying structures. In this thesis we provide a formal foundation to resolve the problem of extracting schemas for evolving collections. The results of this problem help us to adopt path extraction to build efficient queries. In practical terms, our first contribution is to introduce a dictionary to track all changes in terms of documents structures within a collection. For each path from the collection, the dictionary maps paths to all their corresponding paths in all structures present in the collection. Due to structural heterogeneity that we consider in this thesis,

attributes could refer to the same kind of information but at different locations inside the documents. Thus, we build our dictionary to track also partial paths. Hence, entries of the dictionary could be any part of a paths inside the documents with all their corresponding representation in other documents. This contribution is motivated by research problem 1. In our second contribution, we introduce the concept of dictionary, the process to build and maintain it. We employ several synthetic datasets composed of collections describing films to evaluate the time required to extract paths on varying structures of documents, e.g., in collections with up to 5,000 distinct structures. Furthermore, we ran experiments to study the time required to track all changes in terms of documents structures as result of the execution of different manipulation operations. We dedicate Chapter 3 to introduce all formal foundations related to our contribution.

- besides the solutions offered in the literature to provide uniform access to heterogeneous collections of documents using whether relational views built on top of unified schemas or defining new querying languages, we built our solution on top of the original underlying structures of documents and based on the queries expressiveness power of most document stores. In practical terms, in our contribution we rely on using a minimum closed kernel composed of unary, i.e., select, project, unnest and aggregate operators, and binary, i.e., lookup, operators to enable schema-independent querying for heterogeneous documents in NoSQL document stores. All those operators are defined in the nested relational algebra (NRA) which are compatible with the expressiveness of document stores. Therefore, to overcome the heterogeneity in documents, we introduce an automatic query reformulation via a set of rules that reformulate most document store operators. Furthermore, we support queries formulated over partial paths whereas most document stores can run only queries formulated over absolute paths. The query reformulation is performed each time a query executed and thus we guarantee that each reformulated query contains all required absolute paths which are present at the collection in its latest structural status. We conducted a set of experiments over several synthetic datasets and using several workloads. We studied also the time required to reformulate queries and their corresponding execution time. This contribution represents the main contribution of this present thesis. Hence, it is motivated by research problem 2. Therefore, it is introduced and formalized in Chapter 4.

The dictionary construction process it is not only exclusive to the one presented in this thesis. We built our query reformulation process in a way that is not tightly coupled to the dictionary construction. The coupling of queries to the

dictionary resides only on the use of keys, i.e., paths, existing in the dictionary. Thus, if an administrator manually built the dictionary or the dictionary contains keys which are not even existing paths, our query reformulation rules are still valid since the formal definition of the dictionary consists of set of keys, in our work keys refer to paths which could be partial or absolute, associated to absolute paths. The choice of the dictionary keys is subject to the requirements of the end users.

1.4 Research Overview

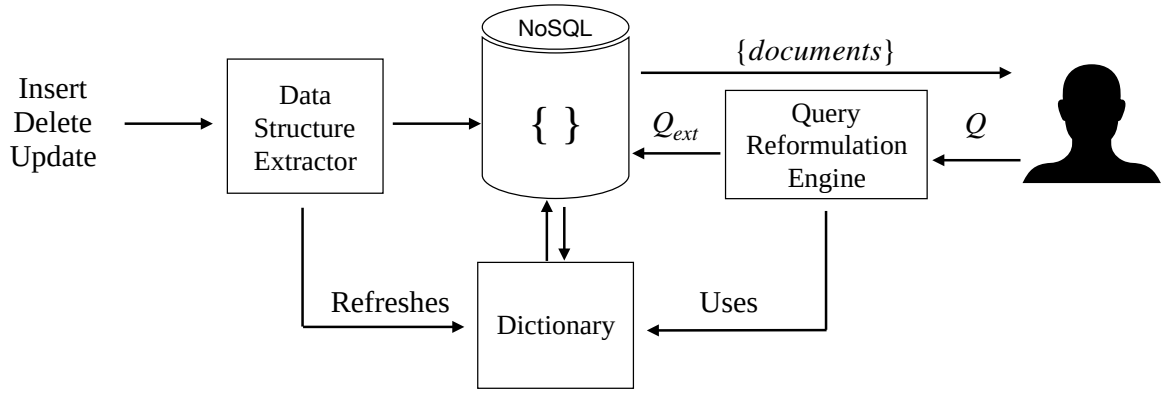

Figure 1.3: EasyQ architecture: data structure extractor (left part) and query reformulation engine (right part).

Figure 1.3 provides a high-level illustration of the architecture of our system called *EasyQ* with its two main components: the dictionary as response to the research problem 1 and the query reformulation engine as a solution to the research problem 2. Moreover, Figure 1.3 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 modified or changed 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 from the dictionary keys (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.

1.5 Manuscript Outline

The remainder of this thesis is organised as follows.

Chapter 2 is dedicated to review the most relevant work from the literature. First, we present a description of the fundamental concepts required for the understanding of our work. Furthermore, we introduce the different solutions to extract structures from collections of heterogeneous documents. Therein, we present the different contributions to enable schema-independent querying for multi-structured collection of documents. Finally, we compare our contribution with respect to state-of-the-art work based on a set of criteria that ined through this chapter.

Chapter 3 discusses the requirements of the research problem 1 targeted to extract paths from multi-structured collections and track all structural changes. This chapter presents formal foundations to define the document data model, the concept of paths and the dictionary. Furthermore, it introduces the process of automating the reformulating classical manipulation operators (insert, delete and update queries) in order to update the dictionary according to the different structural changes made in the collection.

Chapter 4 presents our 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. 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.

Chapter 5 corresponds to the evaluation of our main contributions. In this chapter we present the process of generating our synthetic datasets (available online[2]). Furthermore, we introduce the different workloads that we use to evaluate the performances of our two main contributions. Later on, we draw all experiments results related to dictionary construction and manipulation on varying structures. Therein, we present the results related to executing our schema-independent querying. Furthermore, we introduce three execution contexts to compare results of our contribution with respect to two other execution contexts.

Chapter 6 contains conclusions and some directions for future work.

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

This thesis is financed and conducted in the frame of the neOCampus [3] project. In this project we introduced a novel mechanism to collect data generated from the different IoT sensors, e.g., temperature, humidity, or energy, installed on the classrooms in the building U4 or on the Library at the campus of the University of Toulouse-III Paul Sabatier for later exploration and visualisation. Our mechanism models the sensors data and store it into a NoSQL document Stores, i.e., MongoDB. Furthermore, our mechanism allows to integrate ad-hoc data generated from heterogeneous sensors using different structures. Therein, heterogeneous sensors data could be easily accessed via the use of our API. We address the heterogeneity using a set of automatic reformulation rules that we define in this thesis. Figure 1.4 illustrates our main contributions to this project.

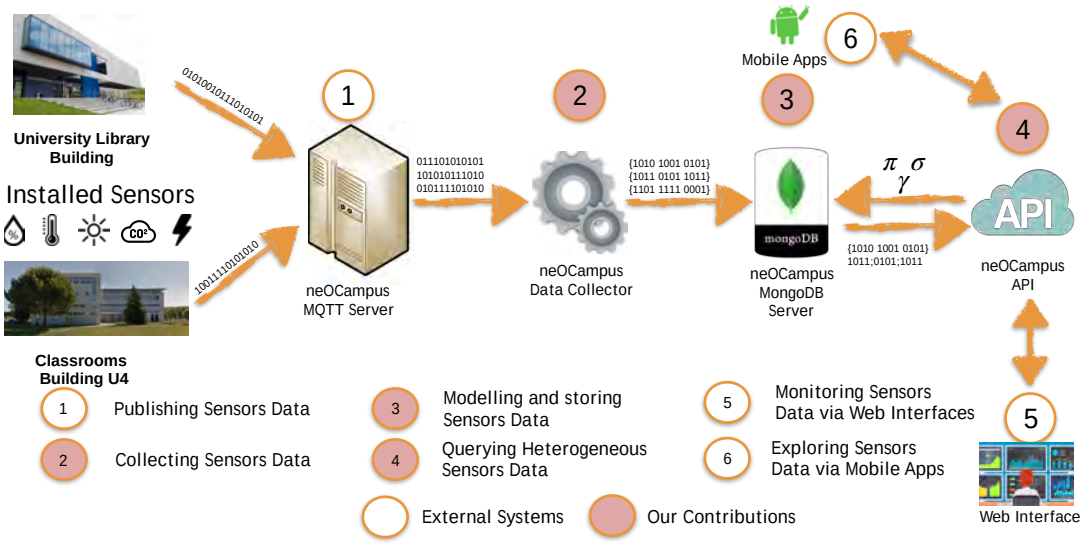

Figure 1.4: Contributions to the neOCampus project.

There follows a list of publications published in the course of this thesis, including the ones that compose the thesis.

- Journal Paper
  - (Ben Hamadou et al., 2019b) Hamdi Ben Hamadou, Faiza Ghozzi, andré Péninou, Olivier Teste : "Schema-independent Querying for Heterogeneous Collections in NoSQL Document Stores" Information Systems, ELSEVIER Volume 85, p. 48-67, 2019. *This paper is presented in particular in Chapter 4 and Chapter 5. It defines the closed kernel of operators, composed of select-project-aggregate-unnest-lookup operators, and their corresponding reformulation rules. Furthermore, it introduces our experimental protocol.*

[3] https://www.irit.fr/neocampus

- International Conferences
  - (Ben Hamadou et al., 2019a) Hamdi Ben Hamadou, Enrico Gallinucci and Matteo Golfarelli: “Answering GPSJ Queries in a Polystore: a Dataspace-Based Approach". In Proceedings 38th International Conference on Conceptual Modeling (ER 2019) Salvador, Brazil, 2019. *This paper is an extension of this present thesis where we showed that the query reformulation rules defined in this thesis could be employed in Polystores context where we overcome further class of heterogeneity.*
  - (El Malki et al., 2018) Mohammed El Malki, Hamdi Ben Hamadou, Max Chevalier, André Pńinou, Olivier Teste : “Querying Heterogeneous Data in Graph-Oriented NoSQL Systems", Data Warehousing and Knowledge Discovery, DaWaK 2018, Ratisbonne, Germany: 289-301. *This paper is an extension of this present thesis where we proved that the query reformulation rules defined in this thesis could be applied to the graph data model, also while considering further class of heterogeneity.*
  - (Ben Hamadou et al., 2018a) Hamdi Ben Hamadou, Faiza Ghozzi, André Péninou, Olivier Teste : “Towards Schema-independent Querying on Document Data Stores", Data Warehousing and OLAP, DOLAP 2018, Vienne, Autriche. *This paper is presented in Chapter 3, Chapter 4. It presents our preliminary ideas regarding the dictionary and query reformulation rules for select and project operators.*
  - (Ben Hamadou et al., 2018b) Hamdi Ben Hamadou, Faiza Ghozzi, André Péninou, Olivier Teste : “Querying Heterogeneous Document Stores", International Conference on Enterprise Information Systems, ICEIS 2018, Madeira, Portugal : 58-68 **Best Student Award**. *This paper is presented in Chapter 3 and Chapter 4. It presents our preliminary ideas regarding the dictionary and query reformulation rules for select, project and aggregate operators.*

- Book Chapter
  - (Ben Hamadou et al., 2019c) Hamdi Ben Hamadou, Faiza Ghozzi, André Péninou, Olivier Teste: Schema-independent Querying and Manipulation for Heterogeneous Collections in NoSQL Document Stores, Springer Lecture Notes in Business Information Processing, volume 363, 2019. *This book chapter is presented in particular in Chapter 3 and Chapter 5. It defines the dictionary and its associated manipulation operation, i.e., insert, update and delete. Furthermore, it presents the evaluation related to the dictionary construction and maintenance.*

- National French Conferences
  - (Ben Hamadou et al., 2018c) Hamdi Ben Hamadou, Faiza Ghozzi, André Péninou, Olivier Teste. "Interrogation de données structurellement hétérogènes dans les bases de données orientées documents", Extraction et Gestion des Conaissances, EGC 2018, Paris, France : 155-166, **Best Academic Paper Award**. *This paper is presented in Chapter 3 and Chapter 4. It presents our preliminary ideas regarding the dictionary and query reformulation rules for select and project operators.*

Chapter 2

Related Literature

Contents

2.1 Introduction

In this thesis, we aim to enable schema-independent querying for heterogeneous collection of documents. For this purpose and to easily understand the remainder of this thesis, we start by introducing the context of this present work. We start by presenting the concept of Big Data (McAfee et al., 2012) and we give details regarding its 3-Vs characteristics, i.e., volume, velocity, and variety. Therein, we give an introduction of the concept of NoSQL stores and their different four data models (Nayak et al., 2013), i.e., key-value, document, column, and graph. Later, we present four classes of work which run to resolve the problem of heterogeneity while querying heterogeneous data. We start first by introducing work proposing the solution of schema integration for this problem and we highlight the limitation of such kind of solution. Second, we describe work addressing the problem by applying physical re-factorisation for the original data model and we show that such solution requires additional efforts. Third, we study work that address the heterogeneity by focusing on the schema inference to understand the different structures and we study their limitations. Forth, we present work resolving this problem with consideration to the querying techniques and we explain how such solutions may require additional efforts from the user. Finally, we draw a summary of all the studied work, and we compare them to our contribution based on some criteria that we define.

2.2 Background

2.2.1 Big Data

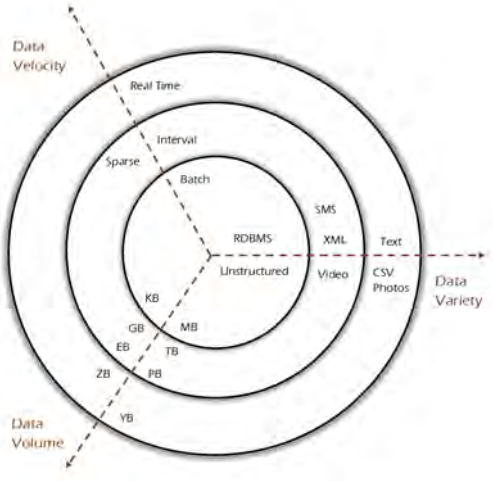

Figure 2.1: Big Data 3-Vs properties, volume, velocity, variety.

Big Data is defined as extremely large datasets that could not be manipulated using conventional systems and solutions (McAfee et al., 2012, Zikopoulos et al., 2011).

Usually, this concept is defined as 3-Vs model, i.e., volume, velocity and variety. These properties are depicted in Figure 2.1 [1] and defined as follows:

- *Volume* refers to the enormous amount of generated data to be processed. Thus, during the last decade, applications such as IoT, web, or social media are used intensively in several domains, e.g., banking, transportation, smart cities, or health, and contribute to the generation of important volumes. For instance, a smart city of 1 million population would generate 200 million GB data per day, going up to 600 ZB by 2020 (Index);

- *Velocity* refers to the speed with which data are being generated to be ingested. Recently, most applications are offering real-time services thus data could be generated every microsecond. For instance, in applications such as social media, e.g., more than 95 million photos and videos are uploaded to *Instagram* every day (O'Donnell, 2018);

- *Variety* refers to the fact that data comes in diverse types of formats, e.g., structured such as relational databases, semi-structured such as XML, or unstructured such as multimedia contents. Furthermore, structures could not be specified beforehand and evolves during time (Kaur and Rani, 2013).

In recent years, Big Data was defined by the 3Vs but now there are more complementary characteristics which are discussed in the community, e.g., Veracity, Value.

2.2.2 NoSQL Stores

To efficiently manage data in Big Data world, NoSQL stores, that stands for Not only SQL, stores become a mainstream solution to store and analyse enormous amount of generated data. Moreover, NoSQL systems run in distributed environments and thus offering high availability. Furthermore, NoSQL systems are well-tailored to scale when there is a need for more storage space for instance. All these capabilities make NoSQL systems becoming an alternative to conventional relational stores while implementing data-intensive applications. The variety characteristic of the Big Data is a naturally support in most NoSQL stores. Hence, the schema-less nature of most NoSQL stores allows the storage of data regardless of their schemas since it is not mandatory to define schemas beforehand (Gómez et al., 2016). Despite conventional relational databases, NoSQL stores offer promising scaling-out capabilities to load the enormous size of generated data (Pokorny, 2013). From the literature we distinguish four mainstream data models for NoSQL stores (Nayak et al., 2013) described as follows:

[1] source: https://www.tutorialscampus.com/tutorials/hadoop/big-data-overview.htm

- **Key-value.** A key-value store is a database designed for storing, retrieving, and managing associative arrays, i.e., set of keys identifying values, which is a data structure grouping data in a dictionary or hash table. Dictionaries store a collection of objects, or records, which in turn have many different fields within them, each containing data. These records are stored and retrieved using a key that uniquely identifies the record and it is used to quickly find the data within the database. Figure 2.2 gives an illustration of the key-value data model. Key-value stores are optimised to deal with records having for each key a single value, e.g., the first record in Figure 2.2. However, a parser is required to retrieve information from a key identifying a record with multiple values, e.g., the second record in Figure 2.2. Furthermore, for instance, Redis (Carlson, 2013), which is an in-memory key-value store, is built to provide high performances to retrieve data using keys but to a small amount of data that fits into the main memory.

Key	Value
Id1	Alex
Id2	Jane, Australia, Student

Figure 2.2: Key-value data model.

- **Document.** Document stores are one of the mainstream NoSQL data models designed to store, retrieve and manage collections of documents of JSON objects. JSON (JavaScript Object Notation) is a format for formatting semi-structured data in human-readable text (Chavalier et al., 2016). Figure 2.3 illustrates the document data model. We notice that the document data model allows the storage of both primitive values, e.g., the value of the attribute *name* in the first document, and object values using nested structures, e.g., the value of the attribute *details* in the second document.

```
{"_id": "id1", "name": "Alex"}
{"_id": "id2",
        "details": {
           "name": "Jane",
           "country": "Australia",
           "occupation": "Student"
         }
}
```

Figure 2.3: Document data model.

Another particularity of this class of stores is the rich query expressiveness that covers most operators (Botoeva et al., 2018), e.g., select-project-aggregate-unnest-join, defined in the Nested Relational Algebra (Korth and Roth, 1987) and usually employed in conventional relational data models. Furthermore, there

is no need to define a structure before loading data. Whereas key-value stores are offering limited storing capabilities, document stores scale well and respond to most Big Data requirements. Systems such as MongoDB (Banker, 2011) succeed to manage large collections of documents in distributed environment. However, it does not offer native support to overcome heterogeneity with structures of documents and users should know the different underlying structures while formulating their queries. For instance, in Figure 2.3, if user formulates her query to project all information related to the attribute *name*, only information from the first document are returned. Despite the presence of the attribute *name* in the second document, the native document query engine could not retrieve it.

- **Column.** A column store is a database that stores data by column whereas conventional relational databases store data by rows (Stonebraker et al., 2005). Thus, such stores access only the data it needs to answer a query rather than scanning and discarding unwanted data in all rows which results in efficient query response time. However, an extensive work is required to define the different structures of tables and their corresponding keys based on the workloads. Furthermore, the query expressiveness of such stores is limited. For instance, Cassandra (Lakshman and Malik, 2010) do not offer native support for queries formulated over columns which are not part of the primary key. Also, a query to retrieve tuples where a given column value is different could not be executed using the native query engine. For instance, it is not possible the user to retrieve all records where the column *country* is equal to *USA* form the column store illustrated in Figure 2.4. This is because only the column *id* compose the table primary key. Column store become best option as the backbone in a system to serve data for common extract, transform, load (ETL) and data visualization tools (Abadi et al., 2009, El Malki, 2016).

Id	**Name**	**Country**	**Occupation**
id1	Alex		
id2	Jane	Australia	Student

Figure 2.4: Column data model.

- **Graph.** A graph store is a type of NoSQL database that uses graph theory (West et al., 1996) to store, manage and query nodes and their corresponding relationships. A graph store is essentially a collection of nodes and edges. Such structures support semantic queries (Lim et al., 2009). Graph store have been used in multiple domains, such as data and knowledge management, recommendation engines, ... Like document stores, there is no need to define structures of nodes or edges beforehand and data could be easily stored regardless of their

structures (Vukotic et al., 2015). However, to query such stores there is a need to employ complex querying language such as Cypher (Francis et al., 2018).

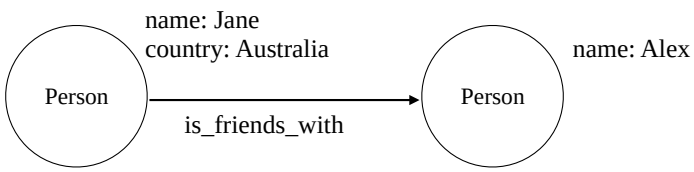

Figure 2.5: Graph data model.

In this thesis, we focus mainly on document stores. We opt for this choice due to the flexibility offered to store data regardless of their structures and the need to overcome the heterogeneity within structures. Furthermore, when compared to the column and key-value stores, the heterogeneity in terms of structures is limited and the schema-less is mainly related to the presence or absence of values for given attributes. In column database, it is required to define a schema for the data beforehand. Both, document and graphs offer the freedom to store data regardless of their schemas. However, more challenges are present in graph stores since the heterogeneity could not only affects the data structures, i.e., structure of information within nodes, but also it could affect edges also since the edges contain information regarding the semantic of the relationships between nodes. However, in addition to our main contributions to overcome the heterogeneity while querying heterogeneous collection of documents, we succeeded to propose some preliminary contributions to address this problem in graph data model. Our results are published in the International Conference on Big Data Analytic and Knowledge Discovery DAWAK 2018 (El Malki et al., 2018). Furthermore, the native query engine in document stores do not offer native support to overcome the heterogeneity. However, it is mandatory for the users to explicitly include all possible paths leading to the same information.

Contexts such as data-lake (Hai et al., 2016), federated database (Sheth and Larson, 1990), data integration, schema matching (Rahm and Bernstein, 2001), and recently, schema-less data support in NoSQL systems (Corbellini et al., 2017) 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 (Ben Hamadou et al., 2018a).

In the reminder of this chapter, we classify state-of-the-art research work based on the solutions proposed for querying multi-structured documents. We start by presenting the first family of work employs different methods of schema matching to resolve

the problem of heterogeneity in structures. Afterwards, we present some work performing materialized structural changes to unify heterogeneous forms of documents. Therein, the third line of work recommends operating queries on a virtual schema derived from the heterogeneous structures and the last recommends querying techniques to overcome the heterogeneity in documents.

2.3 Schema Integration

Current data-intensive applications, e.g., web or IoT, could model the same real-world domain differently since the schemas are independently developed. This difference could be related to the evolution of the application, the integration of data from different sources, etc. Thus, in order to get insights from data generated in diverse structures there is a need to identify relationships between schemas. In this section, we study the most relevant work suggesting schema integration process as an intermediary step to facilitate a query execution process over data having various structures. Thus, this process suggests determining mappings between attributes from heterogeneous schemas to provide integrated access to heterogeneous data using global schema. Furthermore, schema integration employs a set of techniques and approaches to provide the user with a unified structure (Lenzerini, 2002). Usually, this unified structure is called global schema. Therefore, data generated from different sources could be accessed easily regardless of their sources or underlying structures. In practical terms, schema integration involves combining data residing in different sources and providing users with a unified access (Lenzerini, 2002). This process becomes significant in several applications, e.g., commercial when two companies need to merge their databases, or scientific combining results from different bio informatics repositories. It has become the focus of extensive theoretical research work, and numerous open problems remain unsolved.

Several techniques could be applied in order to automate this process. In their survey paper (Rahm and Bernstein, 2001), the authors presented the state-of-the-art techniques used to automate the schema integration process. Matching techniques could be applied to structure-level (Do and Rahm, 2002). Thus, structure-level techniques compute the mapping between structures by analysing the similarity of how attributes appear together in a structure (Shvaiko and Euzenat, 2005). Thus, structure-level matching determines that certain attributes of a given schema semantically correspond to certain attributes of another schema. In (Madhavan et al., 2001) the authors propose an algorithm, i.e., Cupid, that discovers mapping between schema elements using linguistic and structural matching. Hence, such an approach is built to map XML schemas or other structured data where an explicit schema definition is already present whereas data in most NoSQL stores are stored without prior schema definition.

However, current solutions are often very brittle because they only exploit evidence that is present in the two schemas being matched. Thus, in document stores, and due to the lack of additional meta-data annotations of documents, determining the relationship between schemas using only the names of the attributes could result in wrong matches and thus could affect the correctness of the queries evaluation. For instance, it is possible to find a match between two attributes, e.g., an attribute called *location* may refer to two different concepts, i.e., room number, or IP address and (Ventrone, 1991).

Furthermore, matching could be defined between instances (Wang et al., 2004). In practical terms, determining a set of correspondences that identify similar elements in different schemas (Madhavan et al., 2005). The problem of such type of matching is that it is mandatory to perform a comparison between the values of different attributes. In a context such as relation, column stores, it is possible to opt for comparing only a subset of values and thus it is possible to determine matches. However, in contexts such as document or graph stores, and due to the concept schema-less stores, values for a given attribute have no fixed data type, and may vary from document to another same attribute may have different data types (Chasseur et al., 2013, Gallinucci et al., 2018), e.g., the type of an attribute called *age* could be a string or integer in different documents within the same collection, thus performing element matching require to compare all attributes values to others from different documents and structures.

Traditionally, several kinds of heterogeneity are discussed in the literature (Kang and Naughton, 2003): structural heterogeneity refers to diverse possible locations of attributes within a collection due to documents diverse structures, e.g., nested or flat structures and different nesting levels (Ben Hamadou et al., 2018b); syntactic heterogeneity refers to differences in the representation of the attributes, e.g., usage of acronyms, prefix, suffix and special characters due to several naming convention affecting specifically attribute names (Wolski, 1989); finally, semantic heterogeneity may exist when the same field relies on distinct concepts in separate documents (Shvaiko and Euzenat, 2005).

To automatically find matches, diverse tools are employed to address each class of heterogeneity. For instance, the usage of lexical matches helps to handle syntactic heterogeneity (Hall and Dowling, 1980). Lexical matches are usually ensured via the usage of different edit-distance algorithms. Hence, such algorithms represent a way of quantifying how dissimilar two strings are by counting the minimum number of operations required to transform one string into the other (Bille, 2005). Furthermore, thesauruses and dictionaries, e.g., Wordnet (Li et al., 1995), are used to perform semantic matching (Voorhees, 1993). Also, structural matching helps to compare the document structures and to identify commonalities and differences (Bertino et al., 2004).

The main problem of the different matching techniques is in how to determine adequate parameters. Most of the matching functions require to determine a certain threshold that allows to efficiently find matches. However, the parameters work only for the context on which it was initially determined. Therefore, there are no generic matching techniques and parameters that could be applied for any kind of structures or applications.

Schema integration techniques may present certain issues to efficiently deal with the structure of data. Hence, schema integration techniques make it difficult to support legacy applications built on top of original structures of the data before generating a unified one. Therefore, changing the data structure necessitates changing the queries in the application side. Furthermore, this task is required whenever a new common structure is integrated into the collection data.

2.4 Physical Re-factorisation

In this section, we study another alternative to retrieve information from a set of heterogeneous data. This class of work suggests performing physical re-factorisation to facilitate the access to documents having heterogeneous structures. The common strategy of these work consists in changing the underlying document data model into a relational data model. Thus, they rely on the use of conventional relational querying and storing techniques. In practical terms, documents map to relational tables. To achieve this, there is a need to perform customised transformations and to define some rules on how documents should be mapped into relations since there are no standardised solutions to achieve this. Hence, mapping determines possible tables corresponding to a given collection of documents. Furthermore, types could be automatically extracted to define the different columns while defining the relational tables. Such kind of solutions leads to the loss of the flexibility initially guaranteed in the document data model. Therefore, documents are shredded into columns in one or more relational tables. To this end, it is mandatory to perform some physical transformations to map the document data model into the relational one. However, these solutions ensure that semi-structured data can be queried without any prior schema validation or restriction. A mainstream approach widely used while dealing with XML databases, is to partition documents and transform them into relation data model (Amer-Yahia et al., 2004, Böhme and Rahm, 2004, Florescu and Kossmann, 1999). For instance, MonetDB (Idreos et al., 2012) uses specialised data encoding, join methods, and storage for managing documents encoded in XML. In (Amer-Yahia et al., 2004), the authors use the document type definition, i.e., DTD, to flatten documents and map documents into relational tables. However, despite of the advantages of using relational schema and the expressiveness power of relational operators, par-

titioning data into tables by attributes (Florescu and Kossmann, 1999) affects the performance of the relational system. This is due to the need of performing multiple joins to reconstruct any documents.

The particularity of documents encoded in XML is that the documents are usually annotated with information describing the schemas whereas JSON documents are lacking such schema annotations. Thus, such knowledge helps to easily infer the structures and to facilitate the process of mapping documents into relational data model.

With the wide spread use of JSON format (Bourhis et al., 2017) as flexible and extensible document format, and due to the lack of structure annotations, recent work have been introduced to transform JSON into relational data model. In (Chasseur et al., 2013), the authors introduced a system called Argo to manage JSON data using relational stores. They suggest distributing document values across different tables, according to their types. Furthermore, Argo presents a mapping layer for storing and querying JSON data in a relational system with a custom SQL-like query language. The problem with this work is the necessity to learn a custom query language, i.e., Argo/SQL, and to employ a relational store in addition to the document store. In this work also, they suggest flattening the different nested attributes to fit them with the relational data model.

However, in (Tahara et al., 2014), the authors introduced a system, called sinew, enabling to store documents into relational columns and they propose a layer above the database to provides relational views using PostgreSQL (Momjian, 2001) as the underlying RDBMS. Therefore, document encoded in JSON can be queried and managed in relational stores. To achieve this, sinew, adopt a custom serialization format to materialize document attributes into physical columns. Also, PostgreSQL (Momjian, 2001) adopts the same strategy and uses a custom binary serialization for lading documents encoded in JSON into relational stores.

Furthermore, in (DiScala and Abadi, 2016), the authors proposed a normalizing relational data extractor for JSON data. The extractor creates a functional dependency graph representing the different relationships existing within the attributes. However, this work operates over collections of documents sharing a common schema. Thus, it is possible to find different relational tables for each set of schemas present within the collection. Therefore, several tables are generated based on the functional dependencies. To retrieve data, users are required to formulate their queries while considering the necessary join operations to reconstruct information as stored into their original documents.

In Figure 2.6, we illustrate an example of how an information modelled as document could be re-factorized to match with relational one. In this example we note that the attributes are transformed into relational columns. This example is inspired from the

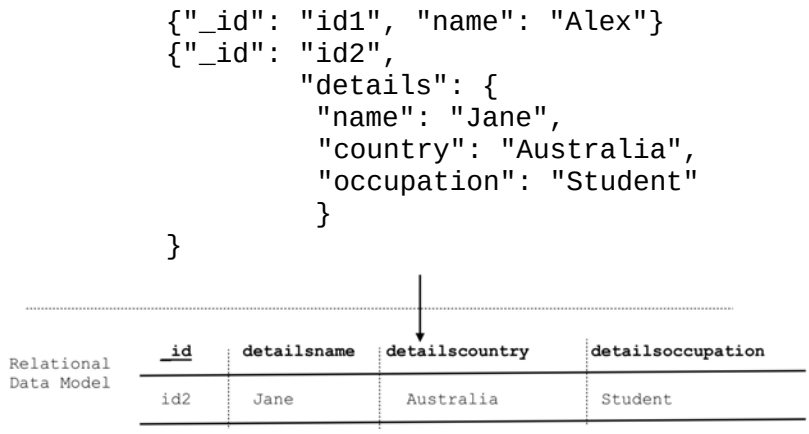

Figure 2.6: Physical re-factorisation of a document data model to relational data model.

work (DiScala and Abadi, 2016).

To retrieve data from the shredded documents, a common strategy consists in formulating relational queries based on relational views built on top of the new data structures. This strategy implies that several physical re-factorisation 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 new generated relational views. Users of these systems have to learn new schemas every time they change the application queries 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.

In addition, the schema-less nature supported by most document stores allow for the same attribute to have diverse data types. Therefore, the physical re-factorisation could lead to several tables for the same information and thus several queries and multiple join operations should be performed to find the expected results.

Finally, adopting custom serialisation to fit document complex attributes, e.g., arrays and objects, into relational could not be very efficient. For instance, Hence, they do not offer fast access to serialised data and thus affects the query performances. For instance, arrays and object require frequent de-serialization before executing the query.

2.5 Schema Inference

In this section, we study the most relevant work conducted to resolve the heterogeneity in documents structures by inferring and unifying their various structures. This problem pushed the community to propose solutions to infer the schemas of document which are implicit in documents. In practical terms, collections are stored regardless of their various structures. To assist the users while formulating their queries.

Several work are proposing schema inference techniques. The idea is to provide users with an overview of the different elements present in the collection of heterogeneous collection of documents (Baazizi et al., 2017, Ruiz et al., 2015).

This family of work was first introduced for inferring structures from semi-structured documents encoded in XML format. These work aim to infer structures using regular expressions rules from the different strings representing element from XML documents to propose a generalized structure (Freydenberger and Kötzing, 2015). Both, JSON and XML are commonly used to encode nested data as documents. However, most of the solutions introduced to infer structures from documents encoded in XML could not be applied to documents encoded in JSON. Furthermore, other efforts were conducted to infer RDF data (Čebirić et al., 2015). The problem with this class of work is none of these approaches is designed to deal with massive datasets whereas current applications are data-intensive, and they are using JSON encoding.

In (Wang et al., 2015) the authors propose a framework to efficiently discover the existence of fields or sub-schemas inside the collection. To this end, the framework is built for managing a schema repository for JSON document stores. The proposed approach relies on a notion of JSON schema called skeleton. Hence, a skeleton is a tree representation describing the structures that frequently appear in a collection of heterogeneous documents. Thus, the skeleton may totally lack some paths that does exist in some of the documents because they do not appear often, and the generation of skeleton will exclude them.

In (Gallinucci et al., 2018) 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, e.g., different type, different location inside documents. Therefore, the authors suggest using mapping to find out the different variation for a given attribute. In order to retrieve information, users should formulate as much query as the number of attributes defined in the mappings for each attribute. However, the problem with such solution is the difficulty to first build the adequate queries since there is no automatic query generation. Second there is a need to perform further treatments to combine the partial results.

In this section, we present the class of work that infer the implicit structures from a heterogeneous collection of documents and provide the user with a high-level illustration regarding all or a subset of structures present inside the collection. This solution could help users to better understand the different underlying structures and to 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. Furthermore, some work does not consider all structures and build an inferred schema on top of most used attributes, for instance, using some probability measures. Thus, queries could result in misleading results. Also, most of the work does not offer automatic support for structures evaluations and it is mandatory to regenerate the inference process which could affect the associated workloads and applications.

We build our schema-independent querying based while getting inspired by this category of work and we provide automatic support for schema evolution. Also, in our contribution, we guarantee that the workload dedicated to fetching data from a heterogeneous collection of documents will not be affected by the evolution in term of structures. We define the concept of the schema of a document because of the lack of a formal specification (Pezoa et al., 2016). The document data model relies on collections whose are usually schema-less. Despite the fact that such flexibility allows providing important capabilities to load huge amounts of semi-structured data regardless of their schema definitions, this flexibility makes it impossible to efficiently formulate complex queries and workloads, users do not have reliable schema information to figure out structural properties to speed up the formulation of correct queries (Baazizi et al., 2017).

2.6 Querying Techniques

From the literature we distinguish some work consisting of proposing query rewriting (Papakonstantinou and Vassalos, 1999) which is a strategy for reformulating an input query into several derivations to overcome heterogeneity. Most research work are designed in the context of the relational databases, 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 (Lin et al., 2017). 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 must be returned to the user.

Furthermore, several systems are offering querying interface where the user could use SQL or SQL-like querying interface to retrieve data from schema-less stores. However, it is required that the user must define the structure of the underlying data before formulating her queries.

To address this problem in the context of document encoded in XML, the system

Pathfinder (Schvaneveldt, 1990) define a processing stack designed to convert from XML and XQuery (Boag et al., 2002) to relational tables and to use SQL queries. This query language inherits the query expressiveness of relational data model from the conventional SQL queries and adds further operators for document data model. Other alternatives for finding different navigational paths which lead to the same nodes are supported by (Boag et al., 2002, Clark et al., 1999). 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 (Florescu and Fourny, 2013), the extension to XQuery designed to deal with large-scale semi-structured data.

Other line of work considers large volume of data and enable querying for documents encoded in JSON. For instance, the work SQL++ (Ong et al., 2014) offers a query language designed to retrieve information from semi-structured data, e.g., documents. Furthermore, we find other work form the literature such as Google Tenzing (Lin et al., 2011), Google Dremel (Melnik et al., 2010), Apache Drill (Hausenblas and Nadeau, 2013), and Apache Spark SQL (Armbrust et al., 2015) propose that user could query data without first defining a schema for the data. In the work Tenzing (Lin et al., 2011), the authors introduce a query mechanism inspired from the SQL querying language and performing in MapReduce systems. To this end they propose to infer relational models from the underlying documents. The limitation of this work is that it is only limited to flat structures. In other words, only documents composed of attributes of primitive types are supported. This is not always valid in the context of documents stores where nested structures is commonly used in current data-intensive applications. In contrast, other solutions such as Dremel (Melnik et al., 2010) and Drill support nested data. We notice also that systems such as Apache Spark SQL (Armbrust et al., 2015) fits the data into main memory using a custom data model called data frames. Hence, a data frame reuses the table structure in which each column contains values of one attribute. In case of heterogeneous structures, data frames are built based on the most used structures and thus leading to miss some elements present in limited number of documents. Furthermore, data frames could be materialised and loaded to optimise querying performances. However, if new structures are inserted into the collection of heterogeneous documents, there is a need to regenerate the data frames and thus changing the columns signature which could affect the existing workloads.

In this thesis, we build our schema-independent querying upon the idea proposed in the context of query reformulation. We believe that the advantage of reformulating queries in ad-hoc fashion is transparent to the already available workloads. Furthermore, in our contribution we focus on generating one query able to overcome the

heterogeneity problem and thus we omit the additional efforts required to recompose results from the execution of several queries.

In the next section we summarise the state-of-the-art work which are related to our work and we discuss them based on set of comparison criteria that we define.

2.7 Summary

In this section, we compare the most relevant work from the literature that propose a solution to facilitate the process of querying heterogeneous data. We present first a table and then we discuss the different criteria. Finally, we position our work with respect to the related literature.

Contribution	Heterogeneity		Querying mechanism	Underlying store	Solution	Data model	Schema evolution support
	Type	Level					
ARGO(Chasseur et al., 2013)	structural	schema	ARGO/SQL	MySQL, Postgres	physical re-factorization	document	manual
Sinew(Tahara et al., 2014)	structural	schema	SQL	Postgres	physical re-factorization	key-value	manual
(DiScala and Abadi, 2016)	semantic	schema	SQL	RDBMS	physical re-factorization	key-value	manual
(Hai et al., 2016)	structural	instance	Keywords queries + SQL	-	data annotation	document	manual
(Baazizi et al., 2017)	structural	schema	-	Distributed DB	schema inference	document	manual
(Ruiz et al., 2015)	structural	schema	-	MongoDB	schema inference	document	manual
SQL++(Ong et al., 2014)	structural	schema	SQL++	RDBMS+NoSQL	query language	relational + document	manual
Spark SQL(Armbrust et al., 2015)	structural	schema	SQL++	RDBMS+NoSQL	query language	relational + document	manual
JSONiq(Florescu and Fourny, 2013)	structural	schema	JSONiq	-	query language	document	manual
XQuery(Boag et al., 2002)	structural	schema	XQuery	-	query language	document	manual
EasyQ(Ben Hamadou et al., 2019c)	**structural**	**schema**	**Aggregation Framework**	**MongoDB**	**query reformulation**	**document**	**automatic**

Table 2.1: Comparative study of the main contributions to querying heterogeneous semi-structured data.

In Table 2.1 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 refactorization 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 each 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?

The majority of the state-of-the-art research concentrates on managing heterogeneity at a structural level. If we consider 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 can 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. We propose to help the user to 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 can reformulate queries with the latest schemas in the collection.

This thesis 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. 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 thesis.

In this chapter, we introduced the main concepts required for the understating of the context and the problematic that we address in this thesis. Therein, we classified related literature to four categories. We introduced some work for each of the category and we compared to our work. Later, we summarised all the work and we draw a comparison table where we compare the different work from the literature with respect to our work. In the reminder of this thesis, we introduce the different formal foundations required to build our main contributions with respect to the assumption that we

introduced in the introduction and the inspiration that we got from the literature.

Chapter 3

Document Data Model Concepts

Contents

During the last decade, NoSQL document stores and schema-less data modelling have emerged as mainstream alternatives to relational modelling for addressing the substantive requirements of current data-intensive applications (Hecht and Jablonski, 2011), e.g., IoT, web, social media and logs. Hence, to build efficient application and to take the most profit from document stores, it is important to start first by understanding the key concepts of the document data modelling. Furthermore, the schema-less nature guaranteed by most document stores requires to build an efficient process to extract the underlying structures within a collection of documents to build efficient queries able to retrieve the required information. This process plays an important role to build adequate queries. Document stores querying engine are not well-tailored to overcome heterogeneity in structures. It is up to the user to handle the heterogeneity and to explicitly includes various absolute paths in their queries to extract information of interest.

In the literature several solutions were introduced to overcome this issue. Current techniques recommend flattening documents, e.g., in XML or JSON format, into a relational form (Chasseur et al., 2013, DiScala and Abadi, 2016, Tahara et al., 2014) to overcome the heterogeneity in structures. This process of physical schema transformation requires additional resources, such as an external relational database and more effort to generate new schema every time they change the workload or when new data are inserted, deleted or updated. Furthermore, to deal with the heterogeneity in structures they propose transforming all document schemas to a single common schema and introducing some logical views which leads to a homogeneous collection (Tahara et al., 2014) using schema matching techniques to merge heterogeneous structures (Rahm and Bernstein, 2001).

Meanwhile current trends suggest performing complex physical transformation, or to employ complex schema matching techniques leading to initial structures loss and affecting the support of legacy applications. We advocate the idea of keeping the documents in their original underlying structures. In this thesis, we provide a formal foundation to resolve the problem of extracting schemas for evolving collections. To track all structures in documents, we introduce a dictionary in which each path within a given structure is mapped to all its corresponding possible absolute paths in other structures.

In this chapter, we introduce the formal model of our proposition built on top of the dictionary to trace the heterogeneity in document structures. We first start by introducing the formal definition of the document data model. Therefore, we introduce our definition of the concept of documents and collection. Then, we introduce our definition of the concept of structure which covers also the concept of paths in documents, and collection structures. Later, based on our formal foundations we define the dictionary that we introduce as a solution to track heterogeneity in the collection

of documents without performing any changes to the underlying document structures. Finally, we define the set of operations, i.e., insert, delete, update, required to maintain the collection schema and the dictionary with the recent structures of an evolving collection of documents.

3.1 Document and Collection Data Model

In this section, we introduce the key concept used in the defintion of the document data model. Therefore, we present formal definition for the concept of collection and document in document data model illustrated with examples.

3.1.1 Collection

In document stores, data is stored and organized as a collection of documents which are simply a grouping of documents (Cattell, 2011). The concept of the collection is like its counterpart table in Relational Database Management systems (RDBMS). A collection can store documents which are not sharing similar structures. This is possible because of the schema-less support of most NoSQL stores. However, the value of documents is encoded in JSON [1] or XML (Consortium et al., 2006) format. Many applications can thus model data regardless of prefixed schemas, as data can be nested with different nesting levels and it is always query-able.

In the following, we introduce the formal definition for the concept of collection in document stores.

Definition 3.1. 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.

3.1.2 Document

The key concept in document data modelling is the document. Most document stores use documents as basic units for data storage as well as for queries. Documents are well-tailored to store large volume of information (Imam et al., 2018). Furthermore, documents are composed of set of fields where any number of fields could belong to the documents without adding same fields with null values to the other documents in a collection. Therefore, the number of attributes in two documents from the one collection could be not the same. Compared to relational databases, empty columns

[1] https://www.json.org/

contain null value by default. The document concept in document stores correspond to records (Kaur and Rani, 2013) in relational ones.

In the following we introduce the formal definition for the document as a key-value pair where the key identifies the document within the collection and the value refers to the document value which could be atomic or complex.

Definition 3.2. 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 d_i in the collection C,
- 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 four following 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 two following forms:

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

In the definition of a document, the document value v_{d_i} is a value $v_{d_i} = \{a_{d_{i,j}} : v_{d_{i,j}}\}$ where the values $v_{d_{i,j}}$ are defined as the generic values which could be atomic, complex or array. To cope with nested documents and navigate through schemas, we adopt classical navigational path notations (Bourhis et al., 2017, Hidders et al., 2017). For instance, to access the value $v_{i,j}$ from a document value defined as $v_{d_i} = \{a_i : \{a_{i,j} : v_{i,j}\}\}$, the corresponding classical navigational path is expressed as $a_i.a_{i,j}$.

Example. Figure 3.1 presents a document identified with the attribute *_id* and its document value is composed of 2 attributes. We distinguish one simple attributes, i.e., *title*, and a complex attribute, i.e., *info*. The latter is composed of three simple attributes, i.e., *year*, *country* and *link*, and three complex attributes: *i)* the attribute *genres* is an array of Strings, *ii)* the attribute *people* which is an object attribute composed of the two object attributes *director*, *lead_actor* in which are nested two simple attributes *first_name* and *last_name* and the array attribute *actors*, and *iii)* the object attribute *ranking* in which is nested the simple attribute *score*.

```
{  "_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}
   }
}
```

Figure 3.1: Illustrative example of a document describing a film.

3.2 Document and Collection Schemas

3.2.1 Paths

In documents, data is present at different nesting levels. The name of an attribute with information of interest is not enough to fetch data when compared to its analogous column name in relational data model. To retrieve required information from documents, it is mandatory use paths expressed in dot notation as in URLs for navigating through the hierarchical structure of a document (Clark et al., 1999).

In the following we introduce the definition for the concept of path, and we distinguish between the different forms of paths, i.e., absolute path, partial path and leaf node.

Definition 3.3. 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. In the event of an array value, the path is expressed same ways as value and ends with the index of each value inside the array. In all cases, the path from the root to any atomic or complex document value in v_{d_i} is called 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*. Finally, leaf node attributes are considered as paths too since they respond to the *partial* path definition.

Example. Figure 3.2 presents snippets from four documents with focus on the attributes *score*. In this example the path *ranking.score* in document (a) represents an absolute path to reach the information referenced by the attribute *score*. This path is composed of the complex attribute *ranking* expressed in dot concatenation format with the simple attribute *score*. However, the same path, *ranking.score* in documents (b, c, d) represents partial paths. Also, we could notice that *others.ranking* in document (c) is another example of a partial path. Furthermore, the path *score* which is a leaf node is considered as partial paths in all documents of the collection (C). In the event of array value, e.g., *ranking*, the path *ranking.1* in document (a) represents an absolute path to retrieve the value *Drama* from this array value.

```
{  "_id":1,
    ....
   "genres":["Drama", "Sport"],
   "ranking":{ "score":8.1}
}
```

(a)

```
{  "_id":2,
   "info":{
    ....
      "ranking":{ "score":7.2}
   }
}
```

(b)

```
{  "_id":3,
   "film":{
      ....
      "others":{
         "ranking":{ "score":8.1}
      }
   }
}
```

(c)

```
{  "_id":4,
....
   "classification":{
      "ranking":{"score":7.2},
....
   }
}
```

(d)

Figure 3.2: Snippets from the collection (C).

3.2.2 Document Schema

In this part, we define the concept of document schema because the lack of a formal specification (Pezoa et al., 2016). Thus, in our definition, we rely on the concept of document paths and we define the document schema as the set of all absolute paths existing in this document.

Definition 3.4. Document Schema

The document schema S_{d_i} inferred from the document value v_{d_i} of a document d_i, is defined as:

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

where, $\forall j \in [1..N_i]$, 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 document 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:

for all attribute $a_{d_i,j} : v_{d_i,j}$:

- if $v_{d_i,j}$ is atomic, $S_{d_i} = S_{d_i} \cup \{a_{d_i,j}\}$ where $a_{d_i,j}$ is the path leading to the value $v_{d_i,j}$;
- if $v_{d_i,j}$ is an object value, $S_{d_i} = S_{d_i} \cup \{a_{d_i,j}\} \cup \{\cup_{p \in s_{d_i,j}} a_{d_i,j}.p\}$ where $s_{d_i,j}$ is the document schema of $v_{d_i,j}$ and $a_{d_i,j}.p$ is the path composed of the complex attribute $a_{d_i,j}$ dot concatenated with the path p of $s_{d_i,j}$;
- if $v_{d_i,j}$ is an array value, $S_{d_i} = S_{d_i} \cup \{a_{d_i,j}\} \cup \{ \cup_{k=1}^{m_j} \Big(\{ a_{d_i,j}.k\} \cup \{\cup_{p \in s_{d_i,j,k}} a_{d_i,j}.k.p\} \Big) \}$ where $s_{d_i,j,k}$ is the document schema of the k^{th} value in the array $v_{d_i,j}$, $a_{d_i,j}.k$ is the path leading to the k^{th} entry from the array value $v_{d_i,j}$ composed of the array attribute $a_{d_i,j}$ composed of the array attribute $a_{d_i,j}$ dot concatenated with the index k, $a_{d_i,j}.k.p$ is the path leading to the k^{th} entry from the array value $v_{d_i,j}$ composed of the path leading to the k^{th} entry from the array dot concatenated with the path p of $s_{d_i,j,k}$; we adopt this notation from (Hidders et al., 2017).

Example. The document schema for the document from Figure 3.1 is as follows:

$S_b =$

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

3.2.3 Collection Schema

After defining the concept of document schema, we introduce now a generic definition to cover the collection schema.

Definition 3.5. Collection Schema

The schema S_C inferred from a collection C is the set of all absolute paths defined in document schemas extracted from each document in the collection C:

$$S_C = \{(p_1, n_1), \ldots, (p_{n_c}, n_{n_c})\}$$

where p_l is an absolute path in the collection and n_l is its number of occurrence within the collection. We refer to the number of occurrence n_l of a path $p_l \in S_C$ as $n_l = S_C(p_l)$. We introduce the number of occurrence n_l to ensure that all paths inside S_C are present in at least one document, $\forall l \in |S_C|, n_l > 0$.

Algorithm 1: Algorithm to construct collection schemas from document Paths.

```
   Input  : C
1  S_C ← ∅
2  foreach d_i ∈ C do
3  |  foreach p_j ∈ S_{d_i} do
4  |  |  if p_j ∉ S_C then
5  |  |  |  S_C(p_j) ← S_C(p_j) + 1
6  |  |  end
7  |  |  else
8  |  |  |  S_C ← S_C ∪ {(p_j, 1)}
9  |  |  end
10 |  end
11 end
12 return S_C
```

In Algorithm 1, the collection schemas S_C is constructed from each path p_j from each document schema d_i, Line $2-3$. In case of an existing absolute path in the document schema, the algorithm increments its number of occurrence by one, Line $4-6$. Otherwise, the algorithm adds new entry to the collection schemas, Line $7-9$. We define the function $generateSchema(C)$ to refer to the Algorithm 1. This function takes as input the collection C and returns the collection schema S_C.

Example.

Figure 3.3 presents a snippet from a collection of documents describing movies. The corresponding collection schemas for this collection is as follows:

$S_C =$

```
{title: 2,
genres.1:1,
genres.2:1,
director:1,
director.first_name:1,
director.last_name:1,
lead_actor.first_name:1,
lead_actor.last_name:1,
ranking:1,
ranking.score:1,
info:1,
info.people:1,
info.people.director:1,
info.people.director.first_name:1,
info.people.director.last_name:1,
info.ranking:1,
film:1,
film.title:1,
```

```
film.details:1,
film.details.director:1,
film.details.director.first_name:1,
film.details.director.last_name:1,
film.details.personas:1,
film.details.personas.lead_actor:1,
film.details.personas.lead_actor.first_name:1,
film.details.personas.lead_actor.last_name:1,
others:1,
others.ranking:1,
others.ranking.score:1,
description:1
```

```
description.title:1,
description.director:1,
description.director.first_name:1,
description.director.last_name:1,
description.stars:1,
description.stars.lead_actor:1,
description.stars.lead_actor.first_name:1,
description.stars.lead_actor.last_name:1,
classification:1,
classification.ranking:1,
classification.ranking.score:1
}
```

```
{  "_id":1,
   "title":"Million Dollar Baby",
   ...
   "genres":["Drama", "Sport"],
   "director":{ "first_name":"Clint",
      "last_name":"Eastwood"
   },
   "lead_actor":{ "first_name":"Clint",
      "last_name":"Eastwood"
   },
   ...
   "ranking":{ "score":8.1
   }
}
```

(a)

```
{  "_id":2,
   "title":"In the Line of Fire",
   "info":{
     ...
      "people":{
         "director":{ "first_name":"Clint",
            "last_name":"Eastwood"
         },
         "lead_actor":{ "first_name":"Clint",
            "last_name":"Eastwood"
         },
         ...
      },
      "ranking":{ "score":7.2
      }
   }
}
```

(b)

```
{  "_id":3,
   "film":{
      "title":"Gran Torino",
      ...
      "details":{
        ...
         "director":{ "first_name":"Clint",
            "last_name":"Eastwood"
         },
         "personas":{
            "lead_actor":{ "first_name":"Clint",
               "last_name":"Eastwood"
            },
        ...
         }
      },
      "others":{
         "ranking":{ "score":8.1
         }
      }
   }
}
```

(c)

```
{  "_id":4,
   "description":{
      "title":"The Good, the Bad and the Ugly",
      ...
      "director":{ "first_name":"Sergio",
         "last_name":"Leone"
      },
      "stars":{
         "lead_actor":{ "first_name":"Clint",
            "last_name":"Eastwood"
         },
         ...
      }
   },
   "classification":{
      "ranking":{"score":7.2
      },
    ...
   }
}
```

(d)

Figure 3.3: Snippets of illustrative example of a collection (C) with four documents describing films.

The document schema concept provides only information for a document structure. Furthermore, the concept of collection schema provides a global view regarding all paths present in all documents within a collection. Also, the paths in those concepts

are restricted to only absolute paths. In the next section, we extend those definitions and we introduce the concept of a dictionary in which we map each path regardless of its nature, e.g., partial or absolute path, with all absolute paths leading to the same piece of information in other document schemas.

3.3 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 (Ben Hamadou et al., 2019b,c).

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. The goal of introducing the concept of partial paths is to extend the query expressiveness of most document stores. Thus, a user will be able to formulate queries over partial paths whereas current practices require that queries are only formulated over absolute paths. Let us notice that dictionary keys will become the basic element on which queries could be formulated.

3.3.1 Document Paths

In the following, we introduce the formal definition of the document paths.

Definition 3.6. Document Paths

We define $P_{d_i} = \{p_{d_i}\}$ as the set of all existing paths in a document d_i: absolute paths as well as 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 each document $d_i = (k_{d_i}, v_{d_i})$, where $v_{d_i} = \{a_{d_i,1} : v_{d_i,1}, \ \ldots, \ a_{d_i,n_i} : v_{d_i,n_i}\}$

- 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 k^{th} value of $v_{d_i,j}$ (document paths for $v_{d_i,j}$).

Since sets contain paths considered as values, 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 Figure 3.1 is as follows:

$P_b =$

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

3.3.2 Collection Paths

After defining the concept of document paths, we introduce now a generic definition to cover the collection paths.

Definition 3.7. Collection Paths

The set of all existing paths (absolute paths and partial paths) in a collection C:

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

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

3.3.3 Dictionary

The main purpose behind introducing document and collection paths is to construct the dictionary.

Definition 3.8. 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, $n_k = |\triangle_{p_k}^C|$.

We refer to the set of absolute paths $\triangle_{p_k}^{C}$ leading to the path p_k in the collection C as $\triangle_{p_k}^{C} = Dict_C(p_k)$.

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. The Algorithm 2 presents the required steps to construct the dictionary. This Algorithm takes as input the collection schema S_C, and collection paths P_C. Later on, it starts to find out for each path in collection paths P_C, Line 2, the set of its corresponding absolute paths from the collection schema S_C, Lines $5-11$. Finally, the algorithm appends the dictionary with a new entry mapping a path with all its absolute paths extracted from other structures of documents, Line 12. In the reminder of this thesis we use the function *generateDict(C')* to use the Algorithm 2 to construct a new dictionary for a collection C' given as input. This function returns the dictionary $Dict_{C'}$ and the collection schema S'_C.

Algorithm 2: Dictionary construction algorithm.

```
   Input : S_C, P_C      // Collection Schema S_C and Collection Paths P_C
 1 dict_C ← ∅                                              // Initialisation
 2 foreach p_k ∈ P_C       // for each path absolute or partial p_k in P_C
 3  do
 4     △^C_{p_k} ← ∅
 5     foreach p_j ∈ S_C                  // for each absolute path p_j in S_C
 6      do
 7        if p_j = p_l.p_k    // if the path p_j is an absolute path expressed
            as a concatenation of a path p_l and the path p_k
 8        then
 9           △^C_{p_k} ← △^C_{p_k} ∪ {p_j}        // add the path p_j to the value △^C_{p_k}
10        end
11     end
12     dict_C ← dict_C ∪ {(p_k, △^C_{p_k})}          // append the dictionary with a
        key-value pair where p_k is a dictionary entry, and △^C_{p_k} is the
        value of the entry p_k
13 end
14 return dict_C
```

Example. For example, if we build the dictionary from q collection composed of document in Figure 3.1, the dictionary keys will contain *title* and *info.people*, but also *info.people.director*, *people.director*, *people*, *director* and so on as explained in the example introduced in section 3.3.1. In the following example, we present the following dictionary entries from the collection (C) in Figure 3.3

- the absolute path *film.title* from document (c);
- the leaf node *score* from documents (a,b,c,d);
- the partial path *people.director* from document (b);
- the partial path ending with leaf node *director.first_name* from documents (b,c,d).

The corresponding values for these dictionary entries are as follows:

- `film.title:[film.title]`
- `score:[ranking.score, film.others.ranking.score, info.ranking.score, classification.ranking.score]`
- `people.director:[info.people.director]`
- `director.first_name:[director.first_name, film.details.director.first_name, info.people.director.first_name, description.director.first_name]`

We introduced the concept of the dictionary after defining all concepts related to schema and paths for documents and collection. The dictionary provides a full coverage regarding the presences of the paths and their distribution across diverse structures of documents in the collection. The information related to the structures is static, i.e., this information only refers to the exact structures composing the document of a collection now when the dictionary is created. Thus, it is required to update the dictionary when structures of documents evolve during the time.

3.4 Dictionary Maintenance

In this section, we introduce an automatic mechanism that keeps the dictionary updated with the latest structures within a collection of documents. The main idea behind this process is to track every manipulations, e.g., insert, update or delete, that occurs to a collection and to simultaneously update the affected paths by those operations (Ben Hamadou et al., 2019c). Therefore, in case of document with new structures, the mechanism; *i)* adds new keys to the dictionary with all new paths from new structures, *ii)* updates the value of existing keys with new absolute paths from new structures. Moreover, in case of update operation, the mechanism updates the dictionary entries corresponding to the paths affected by the update operation. Finally, in case of delete operation the mechanism removes *i)* keys from the dictionary and their corresponding absolute paths in case of obsolete dictionary keys. *ii)* obsolete absolute paths leading to existing dictionary keys.

Collection manipulation operators are used to insert, delete, and modify (update) documents in a collection. Storing data in their original schema in classical document

stores, we use classical manipulation operators of document stores. Since these operations may lead to changes in schemas of documents, we add to these operators a simultaneous operation to update the collection schema and the dictionary accordingly.

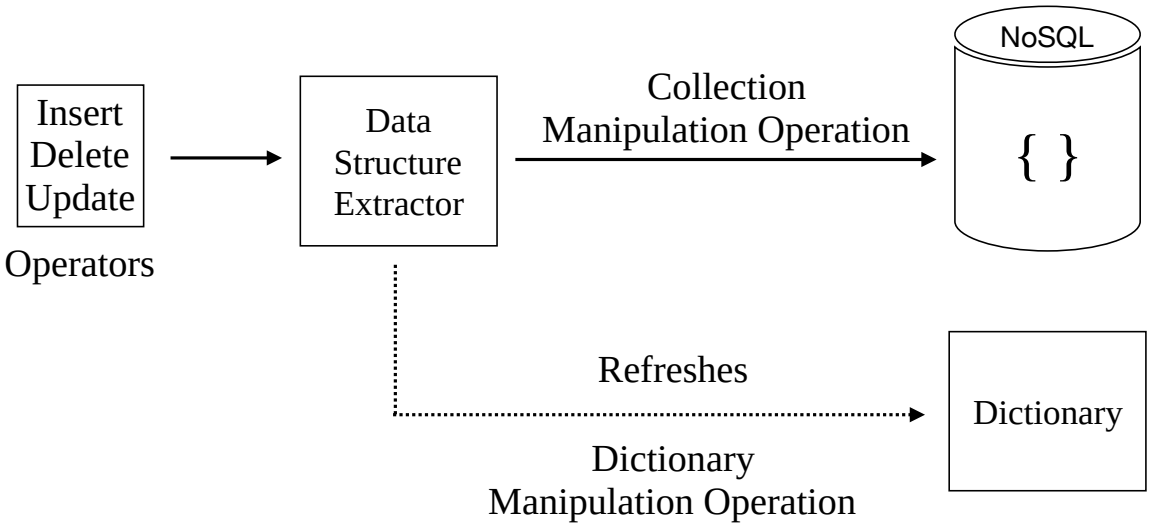

Figure 3.4: Collection manipulation process.

Figure 3.4 shows the process of executing a manipulation operation on a collection. We denote Φ as the insert operator, Ψ as the delete operator and Θ as the update operator. We define any collection manipulation operator as the computation of two pseudo-collections C_{old} and C_{new}. The collection C_{old} refers to the set of documents affected by the execution of the manipulation operation making those documents obsolete and it is mandatory to remove them from the collection C. Conversely, the collection C_{new} refers to the set of new documents with their possible new structure to be inserted in the collection C or possibly replacing documents of the collection C_{old} from the collection C after executing the manipulation operation. Thus, we formally represent the result of any collection operator as follows:

$$C \leftarrow C \setminus C_{old} \cup C_{new}$$

where C is the collection to manipulate, C_{old} the set of documents to remove from C and C_{new} the set of documents to add to C. We provide examples of C_{new} and C_{old} for each manipulation operation in the next sections.

Algorithm 3 presents the different states of both collections C_{old} and C_{new} for each manipulation operation. Line $2-7$ presents the case of insert operation, i.e., Φ. In this case, the collection C_{new} refers to the set of new documents to insert in the collection C. Line $8-13$ presents the case of delete operation, i.e., Ψ. In this case, the collection C_{old} refers to the set of obsolete documents to remove from the collection C. Finally, Line $14-19$ presents the case of update operation, i.e., Θ. In this case, the collection C_{new} refers to the set of documents after updating their status, C_{old} contains the document in their initial status before executing the update

Algorithm 3: Generic manipulation operation.

```
Input : C
switch operator do
    case Φ;                                              // case of Insert
     do
        C_new holds the set of new documents to insert in the collection
        C ← C ∪ C_new
    end
    case Ψ;                                              // case of Delete
     do
        C_old holds the documents to delete from the collection
        C ← C \ C_old
    end
    case Θ;                                              // case of Update
     do
        C_old holds the documents to be updated in their initial state
        C_new holds the set of documents to be updated after applying the
         update
        C ← C \ C_old ∪ C_new
    end
end
return C
```

operation over the collection C. We define the function $Insert(Dict_C, C_{new})$ to the use the Algorithm 4. This function takes as input the initial dictionary to update $Dict_C$, the set of documents to insert C_{new} and it returns the $Dict_C$ after refreshing it.

In the remaining of this section, we introduce the different steps to update the dictionary for each of the manipulation operators.

3.4.1 Insert Operation

In this part, we define the process of updating the dictionary while new documents are inserted in the collection. This process consists on updating the dictionary entries in case of inserting documents with new structures. Furthermore, this process update the collection schemas by adding new possible absolute paths with their corresponding number of occurrence or updating the number of occurrence of existing absolute paths. Therefore, the dictionary entries are simultaneously updated with the newly inserted absolute paths. Furthermore, new entries are added to the dictionary that could be used later during query reformulation (Ben Hamadou et al., 2019c).

Definition 3.9. Dictionary update on insert operation

The execution of this operator is automatically executed whenever new documents are inserted into the collection. We denote the insert operation as:

$$\Phi(C), \text{ where } C_{new} \neq \varnothing, C_{old} = \varnothing$$

Algorithm 4: Dictionary update on insert operation.

```
   Input  : Dict_C, C_new
 1 S_{C_new} ← generateSchema(C_new)              // Same way as Algorithm 1
 2 Dict_{C_new} ← generateDict(C_new)             // Same way as Algorithm 2
 3 foreach p'_k ∈ Dict_{C_new} do
 4 |   if p'_k ∈ Dict_C then
 5 |   |   △^C_{p'_k} ← △^C_{p'_k} ∪ △^{C_new}_{p'_k}
 6 |   end
 7 |   else
 8 |   |   Dict_C ← Dict_C ∪ {(p'_k, △^{C_new}_{p'_k})}
 9 |   end
10 end
11 foreach p' ∈ S_{C_new}      // updating the number of occurrence for each
    absolute path in the collection
12  do
13 |   if p' ∈ S_C then
14 |   |   S_C(p') ← S_C(p') + S_{C_new}(p')
15 |   end
16 |   else
17 |   |   S_C ← S_C ∪ {(p', S_{C_new}(p'))}
18 |   end
19 end
20 return Dict_C, S_C
```

The goal is to update the dictionary $Dict_C$ with possible new paths extracted from the collection of new documents C_{new}. We describe this process as follows:

- adding new entries in the dictionary $Dict_C$ (e.g., new paths in documents);
- adding new absolute paths to initial paths existing in $Dict_C$;
- updating the number of documents for each absolute path in S_C.

The insertion of the new collection C_{new} into the collection C requires to update the dictionary $Dict_C$ as follows:

Algorithm 4 starts by generating the dictionary for the collection C_{new}, Line 1, and the collection schema $S_{C_{new}}$, Line 2 . Later, it iterates over each path p'_k in the dictionary $Dict_{new}$, Line 3. If the path p'_k it is already an entry into the dictionary $Dict_C$, the algorithms adds the new absolute paths to the value of the dictionary entry $\triangle^C_{p'_k}$, Lines $4-6$. Otherwise, it creates a new dictionary entry identified by the path p'_k and its associated value $Dict_{C_{new}}(p'_k)$, Lines $7-9$. Finally, the dictionary updates the number of occurrence for each absolute path p', noted $S_C(p')$, added or affected by the insertion operation, Lines $11-19$.

Example. In this example, we execute an insert operation which adds two documents as described in in Figure 3.5 into the collection C from Figure 3.3. Hence, the collection

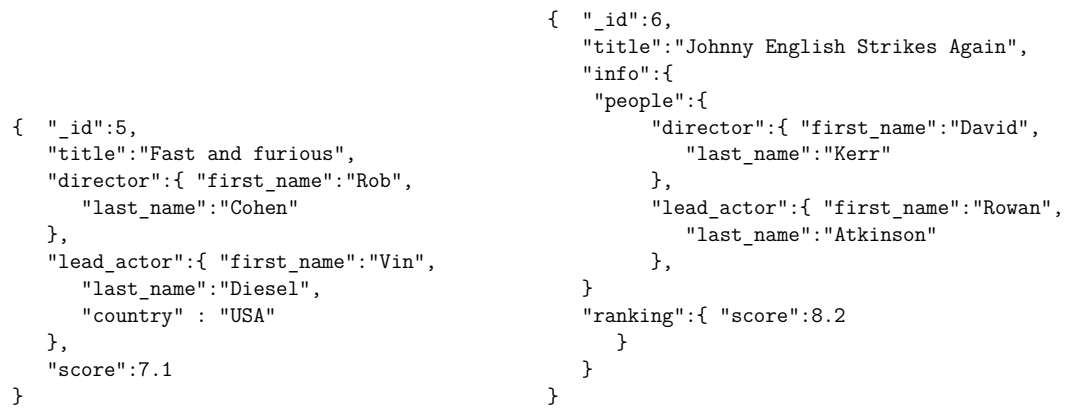

Figure 3.5: Collection (C_{new}) with two documents describing films.

C_{new} to insert in C is composed these two documents. After executing the Algorithm 4, the schema collection S_C from section 3.2.3 becomes

$S_C =$

```
{_id: 6,
title: 4,
director:2,
director.first_name:2,
director.last_name:2,
lead_actor:2,
lead_actor.first_name:2,
lead_actor.last_name:2,
lead_actor.country:1,
ranking:2,
ranking.score:2,
info:2,
info.people:2,
info.people.director:2,
info.people.director.first_name:2,
info.people.director.last_name:2,
info.ranking:2,
info.ranking.score:2
...
}
```

In this example we note the introduction of a new absolute path *lead_actor.country*, extracted from the document of *_id:5*, in the collection schema S_C. Furthermore, the insert operation refreshes the collection schema S_C entries. For instance, the absolute paths *title* was *title:2* before the insert operation, becomes *title:4* since both documents contain the absolute path *title*. In addition to the collection schemas updates, the algorithm 4 updates the dictionary by adding the following entries to the dictionary.

- `country:`
  `[lead_actor.country]`
- `lead_actor.country:`
  `[lead_actor.country]`

Therein, the dictionary updates the entry associated to the key *score* presented in the example in Section 3.3.3 with the new absolute path *score* extracted from the document (a) in Figure 3.5 as follows:

- `score:[score,`
  `ranking.score,`
  `film.others.ranking.score,`
  `info.ranking.score,`
  `classification.ranking.score]`

However, the dictionary entry referenced with the key *director.first_name* presented in Section 3.3.3 stay invariant. This is because these paths are already defined in the dictionary and only the number of their occurrence is affected with this insert operation. Thus, only the schema of the collection is affected where for instance the

entry *info.people.director.first_name* becomes equal to 2 because this path is present in document (b) from Figure 3.5.

3.4.2 Delete Operation

In this part, we describe the different steps required to refresh the dictionary and the collection schemas whenever a delete operation is executed over an existing collection. Therefore, this process removes obsolete absolute paths from the dictionary entries or removes obsolete dictionary entries. Furthermore, it refreshes the collection schemas (Ben Hamadou et al., 2019c).

Definition 3.10. Dictionary update on delete operation

The execution of this operator is automatic whenever a delete operation is executed on the collection C. We denote the dictionary delete operation as:

$$\Psi(C), \text{ where } C_{new} = \varnothing, \text{ and } C_{old} \neq \varnothing$$

The goal is to update $Dict_C$ according to C_{old} by:

- updating the number of documents for each absolute path deleted in S_C;
- deleting unnecessary entries for absolute paths having count equals to 0 from the collection schema S_C;
- updating the dictionary $Dict_C$;
- deleting unnecessary entries in the dictionary $Dict_C$, those having no more absolute paths in the collection to reach them.

Algorithm 5 starts first by generating a dictionary for the set of documents to delete C_{old} and their corresponding collection schema $S_{C_{old}}$, Line $1-2$. Later on, it iterates over the different entries p'_k of the dictionary $Dict_{C_{old}}$ and over each absolute path p' in $\triangle_{p'_k}^{C_{old}}$, Lines $3-4$. Then, if all absolute paths p' are deleted, the algorithm removes the path p' from the dictionary entry $\triangle_{p'_k}^{C}$, Lines $5-6$. Therein, if there are no more absolute paths leading to the path p'_k, the algorithm removes this entry, Line $7-8$. Finally, the dictionary updates the number of occurrences of each absolute path p', noted $S_C(p')$, by subtracting the number of occurrence from the collection schema $S_{C_{old}}$ if not all paths p' from collection C are removed, Lines $13-16$. Otherwise, the algorithm removes the path p' from the collection schema S_C, Lines $17-19$. We define the function $Delete(Dict_C, C_{old})$ to the use the Algorithm 5. This function takes as input the initial dictionary to update $Dict_C$ and the set of documents to remove C_{old} and it returns the $Dict_C$ after refreshing it.

Example. In this example we delete from the collection C the document in Figure 3.6. Therefore, the Algorithm 5 starts by refreshing the collection schema S_C. Thus, the following entries from the collection schema S_C from section 3.2.3 are deleted:

Algorithm 5: Dictionary update on delete operation.

Input : $Dict_C, C_{old}$

1 $S_{C_{old}} \leftarrow generateSchema(C_{old})$; // Same way as Algorithm 1
2 $Dict_{C_{old}} \leftarrow generateDict(C_{old})$; // Same way as Algorithm 2
3 **foreach** $p'_k \in Dict_{C_{old}}$ **do**
4 **foreach** $p^{'} \in \triangle_{p'_k}^{C_{old}}$ **do**
5 **if** $S_C(p') - S_{C_{old}}(p') = 0$ **then**
6 $\triangle_{p'_k}^{C} \leftarrow \triangle_{p'_k}^{C} \setminus \{p'\}$
7 **if** $\triangle_{p'_k}^{C} = \varnothing$ **then**
8 $Dict_C \leftarrow Dict_C \setminus \{(p'_k, \varnothing)\}$
9 **end**
10 **end**
11 **end**
12 **end**
13 **foreach** $p' \in S_{C_{old}}$ **do**
14 **if** $S_C(p') - S_{C_{old}}(p') > 0$ **then**
15 $S_C(p') \leftarrow S_C(p') - S_{C_{old}}(p')$
16 **end**
17 **else**
18 $S_C \leftarrow S_C \setminus \{p'\}$
19 **end**
20 **end**
21 **return** $Dict_C, S_C$

The algorithm 5 removes these entries from the collection schemas since in S_C from section 3.2.3, the corresponding entry for the absolute path *info.people.director.first_name* was *info.people.director.first_name:1*. The delete operation removes the documents, thus the absolute path *info.people.director.first_name* it does not exists any more in the collection schema S_C. Furthermore, the Algorithm 5 removes the following keys:

```
[info                          info.people.director,          info.people.director.last_name,
info.people                    info.people.director.first_name, info.ranking]
```

In addition, if we consider the partial path entry identified by the key *director.first_name* presented in Section 3.3.3, after the execution of the delete operation becomes:

- `director.first_name:[director.first_name, film.details.director.first_name, description.director.first_name]`

3.4.3 Update Operation

In this part, we introduce the different steps required to update the dictionary and the collection schemas while executing a document update operation. The update

```
{  "_id":4,
   "description":{
      "title":"The Good, the Bad and the Ugly",
      ...
      "director":{ "first_name":"Sergio",
         "last_name":"Leone"
      },
      "stars":{
         "lead_actor":{ "first_name":"Clint",
            "last_name":"Eastwood"
         },
         ...
      }
   },
   "classification":{
      "ranking":{"score":7.2
      },
    ...
   }
}
```

Figure 3.6: Document to delete.

operation runs to change document values and even structures (Ben Hamadou et al., 2019c).

Definition 3.11. Dictionary update on update operation

The execution of this operator is automatic whenever an update operation is executed on the collection C. We denote the dictionary remove operation as:

$$\Theta(C), \text{ where } C_{new} \neq \varnothing, C_{old} \neq \varnothing$$

The goal is to update $Dict_C$ according to C_{old} and C_{new}. This update is processed by updating $Dict_C$ from C_{old} as explained for delete and then updating the $Dict_C$ from C_{new} as explained for insert. Let us notice that the processing of update could be somehow reversed, first from C_{new} and then from C_{old}, leading to the same result.

Algorithm 6: Dictionary update on update operation.

Data: $Dict_C$, C_{old} , C_{new}

1 $Delete(Dict_C, C_{old})$; // Same way as Algorithm 5

2 $Insert(Dict_C, C_{new})$; // Same way as Algorithm 4

3 **return** $dict_C$

The Algorithm 6 executes the Algorithm 5 to update the dictionary after deleting the documents from the collection C_{old}, Line 1. Then, it inserts the documents after updating their underlying structures as result of executing the update operation. Hence, the algorithm calls the Algorithm 4, Line 2.

3.5 Conclusion

In this chapter, we introduced the different formal foundations required for the understanding of the two main contributions of this thesis. We started first by defining the

document data model. In order to overcome the heterogeneity, we introduced a set of definition to infer structures from the documents. Thus, we introduced the concept of paths. Paths are used to navigate inside documents starting from the root of the document in case of absolute paths and could be partial paths if they do not start from the root of the document, or they are leaf attribute.

We infer the different possible absolute paths from each document to build the collection schemas. The main idea behind the collection schema is to provide a transparent overview of the different structures of documents within a collection of heterogeneous schemas. The collection schemas trace also the number of documents for each absolute path. This definition helped us later in this chapter to automatically update our schema inference solution with the latest information related to the existing absolute paths within the collection. In order to assist the user while formulating her queries, we introduced the concept of a dictionary. The dictionary represents the set of paths that can be used to formulate relevant queries. Therefore, the user could formulate queries using partial or absolute paths regardless of conventional documents stores. To enable such flexibility, we introduced the definition of document and collection paths. This definition helps to extract from each document within the collection the set of all possible paths. Hence, the user is no more limited to only absolute paths to retrieve information of interest.

In this chapter, we introduced also different mechanisms that automate dictionary maintenance; i.e., whenever a manipulation operation is executed over the collection, we define how to automatically update the dictionary with the existing paths. In case of inserting a document with new structures, we automatically add new entries to the dictionary with all new paths and their associated absolute paths or we update the existing dictionary entries with new possible absolute paths. We perform the same for all of the delete and update operations.

The purpose of this chapter is to offer transparent mechanisms to overcome heterogeneity in documents. Therefore, the introduction of the dictionary helps to tracks all structures within a collection of documents. Furthermore, it provides to the user more flexibility while formulation her queries. It is possible for the user to use partial information of the paths to fetch information of interest. The different algorithms that we introduced to maintain the dictionary help to guarantee that query reformulation that we introduce in the next chapter returns the required information. Hence, they help to remove obsolete paths from the dictionary leading to generate queries that may take more time to execute due to the unnecessary absolute paths included in the query. Moreover, the query reformulation may lack new absolute paths introduced after insert or update operation.

In the next chapter, we introduce a minimal closed kernel of operators inherited from the Nested Relational Algebra (Korth and Roth, 1987) and formalised in the

context of document stores in (Botoeva et al., 2018). In our kernel, we support select-project-unnest-aggregate-join operators. Based on the dictionary, we introduce a set of reformulation rules for most document stores operation that we support. All formal definitions presented in this chapter are necessary to build the automatic process of query formulation to enable schema-independent querying for heterogeneous document stores.

Chapter 4

Schema-independent Querying

Contents

In this chapter, we introduce the different operators that we support to query collections of a heterogeneous collection of documents. In our contribution, we build all our solutions for enabling schema-independent querying using reformulation rules. Thus, we do not introduce a new querying language. In addition to overcoming the structural heterogeneity in document via query reformulation, we extend the query expressiveness power by offering for the user the ability to formulate queries using absolute paths and partial paths whereas most document stores require to formulate queries over absolute paths only. The main advantages of our contribution are that all reformulated queries are compatible with most native document stores querying engines. Therefore, we start first by introducing a minimum closed kernel of operators providing support for a set of unary, i.e., select-project-aggregate-unset, and binary, i.e., join, operators (Ben Hamadou et al., 2019b). Hence, we present for each operator the results when applied over a heterogeneous collection of documents to highlight the limitations of the underlying native querying engine. In practical terms, for each operator, we introduce a set of reformulation rules which employ the different formal definition introduced in the previous chapter, e.g., dictionary, paths, collection schemas and collection paths. Then, we define the set of reformulation rules to overcome heterogeneity in documents structures. In practical terms, for each operator, we introduce a set of reformulation rules which employ the different formal definition introduced in the previous chapter, e.g., dictionary, paths, collection schemas and collection paths. Therein, we present the results of executing the reformulated queries over the same heterogeneous collection of documents.

The reminder of this chapter is as follows. First, we introduce the kernel of operators. Then, we introduce the different reformulation rules for each operator. Finally, we introduce an automatic Algorithm for operator reformulation.

Figure 4.1 presents the collection reference that we employ for the different examples presented in this chapter. We reuse the same collection presented earlier in the first chapter.

4.1 Minimum Closed Kernel of Operators

In this section we define a minimum closed kernel for operators based on the document operators defined in (Botoeva et al., 2018). Later, we introduce the definition of the query.

Definition 4.1. Kernel

The kernel K is a minimal closed set composed of the following operators:

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

```
{  "_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)

Figure 4.1: Illustrative example of a collection (C) with four documents describing films.

The selection, 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.

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:

Definition 4.2. Query

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

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

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 and allow some algebraic manipulations which are helpful in reducing the query complexity. However, such optimisations are out of the scope of this thesis and are subject of future work.

In the next sections, each operator is studied in five steps. We first give the operator definition, based on 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. Since we target that reformulated queries should be evaluated using the classical querying engines, it is necessary that we define the classical evaluation of operators to define the reformulation of operators so that these reformulations are correctly evaluated, particularly when considering missing paths and *null* values.

4.2 Selection Operation

In this section we introduce the selection operator for document stores. We give the definition of the operator and its normal execution over a heterogeneous collection of documents. Later, we define the reformulation rules required to overcome the heterogeneity.

Definition 4.3. Selection

The selection operator is defined as:

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

The selection 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 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} (Ben Hamadou et al., 2018a).

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 3.1.2, $v_k = \{a_{k,1} : v_{k,1}, \ldots, a_{k,n} : v_{k,n}\}$

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:

$$P = \bigwedge \Big(\bigvee p_k \ \omega_k \ v_k \Big)$$

Example. Let us suppose that we want to execute the following selection operator on collection (C) from Figure 4.1:

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

This selection operation only selects movies produced starting from the year *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”}*.

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

Example. In a classical evaluation, the execution of the above-mentioned selection operation returns the following documents:

This query selects only 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}}
}
```


Due to the presence of some partial paths in our query and 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.

4.2.2 Selection Reformulation Rules

The reformulation of the selection 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 for the path p_k. 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:

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

The query reformulation engine start first by extracting the following entries from the dictionary:

- the absolute paths leading to the path *year*, i.e., $\triangle_{year}^{C}$, are equal to *[year, info.year, film.details.year, description.year]*
- the absolute paths leading to the path *director*, i.e., $\triangle_{director}^{C}$, are equal to *[director, info.people.director, film.details.director, description.director]*

Then, it reformulates each condition as follows:

- the condition $\text{year} \geq 2004$ becomes:

$$\text{year} \geq 2004 \vee \text{info.year} \geq 2004 \vee \text{film.details.year} \geq 2004 \vee \text{description.year} \geq 2004$$

- the condition director = {“first_name":“Clint", “last_name":“Eastwood"} becomes:

$$\begin{aligned}&\text{director}=\{\text{“first\_name":“Clint", “last\_name":“Eastwood"}\}\\ \vee\ &\text{info.people.director} = \{\text{“first\_name":“Clint", “last\_name":“Eastwood"}\}\\ \vee\ &\text{film.details.director} = \{\text{“first\_name":“Clint", “last\_name":“Eastwood"}\}\\ \vee\ &\text{description.director} = \{\text{“first\_name":“Clint",“last\_name":“Eastwood"}\}\end{aligned}$$

After applying the reformulation rules, the selection operator becomes:

$$\begin{aligned}&\sigma(\text{year} \geq 2004 \vee \text{info.year} \geq 2004 \vee \text{film.details.}\ \text{year} \geq 2004 \vee \text{description.year} \geq 2004) \bigwedge (\text{director} =\\ &\{\text{“first\_name":“Clint", “last\_name":"Eastwood"}\} \vee \text{info.people.}\\ &\text{director} = \{\text{“first\_name":“Clint", “last\_name":“Eastwood"}\} \vee\\ &\text{film.details.director} = \{\text{“first\_name":“Clint", “last\_name":}\\ &\text{“Eastwood"}\} \vee \text{description.director} = \{\text{“first\_name":“Clint",}\ \ \text{“last\_name":“Eastwood"}\})\end{aligned} \quad (C)$$

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

4.3 Projection

Definition 4.4. 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 (Ben Hamadou et al., 2018a). 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 all documents is changed (Ben Hamadou et al., 2019b).

The sequence of project elements is defined as $E = e_1, \ldots, e_{n_E}$, 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 Figure 4.1:

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

4.3.1 Classical Projection Evaluation

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

- *i)* 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 the path p_j with the value v_{p_j} from d_i ($p_j \in S_{d'_i}$),

  - 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 the path p_j, ($p_j \notin S_{d_{i'}}$);

- *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 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 the path p'_j ($p'_j \notin S_{d_{i'}}$);

- *iii)* $p'_j : [p_1, \ldots, p_m]$ 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 l^{th} 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 $null$;

- *iv)* $p'_j : \beta$, 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]
}
```


Due to the presences of partial paths in our query, 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.

4.3.2 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 $p_j \in P_{C_{in}}$, $e_{j_{ext}} = p_{j,1} \mid \ldots \mid p_{j,n_j} \forall p_{j,k} \in \triangle_{p_j}^{C_{in}}$;
- *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 $p'_j : p_{j,1} \mid \ldots \mid p_{j,n_j},\ \forall p_{j,k} \in \triangle_{p_j}^{C_{in}}$;
- *iii)* $p'_j : [p_1, \ldots, p_m]$, where $[p_1, \ldots, p_m]$ is an array of paths, then each path $p_j \in [p_1, \ldots, p_m]$ is replaced by a "||" combination and $e_{j_{ext}}$ is of the form $p'_j : \left[p_{1,1} \,||\, \ldots \,||\, p_{1,n_1},\ \ldots,\ p_{m,1}||\, \ldots \,||\, p_{m,n_m}\right] \forall p_{j,l} \in \triangle_{p_l}^{C_{in}}$;
- *iv)* $p'_j : \beta$, where β is the boolean expression β, $e_{j_{ext}} = (p'_a\ \omega\ p'_b)$ where $p'_a = p_{a,1} \mid \ldots \mid p_{a,n_a},\ \forall p_{a,l} \in \triangle_{p_a}^{C_{in}}$ and $p'_b = p_{b,1} \mid \ldots \mid p_{b,n_b},\ \forall p_{b,l} \in \triangle_{p_b}^{C_{in}}$.

In the following we introduce the Algorithm 7 to present the automatic process of reformulating the project operator for its different cases.

Algorithm 7 runs over each element in the project operator, Line 2. Then based on the type of the element e_j the dictionary extends it with the corresponding entries from the dictionary, Lines $4-7$ for path element, Lines $8-11$ in case of rename operation, Lines $12-14$ in case of array of elements and Lines $15-19$ in case of boolean comparison of paths.

Algorithm 7: Algorithm for automatic project operator reformulation.

Input : π_E

$E_{ext} \leftarrow \varnothing$ // initialising the set of extended elements from E

foreach $e_j \in E$ // for each element $e_j \in E$

do

  if $e_j = p_j$ *is a path* $(p_j \in P_{C_{in}})$ // e_j takes the form of a path

  then

    $e_{j_{ext}} = p_{j,1} \mid \ldots \mid p_{j,n_{n_j}} \forall p_{j,l} \in \triangle_{p_j}^{C_{in}}$ // generating $p_{j_{ext}}$ using paths from $\triangle_{p_j}^{C_{in}}$

  if $e_j = p'_j : p_j$, $(p_j \in P_{C_{in}})$ // renaming the path p_j to p'_j

  then

    $e_{j_{ext}} = p'_j : p_{j,1} \mid \ldots \mid p_{j,n_j}$, $\forall p_{j,l} \in \triangle_{p_j}^{C_{in}}$ // generating $e_{j_{ext}}$ while renaming paths from $\triangle_{p_j}^{C_{in}}$ to p'_j

  if $e_j = p'_j : [p_1, \ldots, p_{m_j}]$, $\forall l \in [1..m_j], p_l \in S_{C_{in}}$ // new array $[p_1, \ldots, p_{m_j}]$ composed of paths p_l

  then

    $e_{j_{ext}} = p'_j : [p_{1,1}|| \ \ldots || \ p_{1,n_1} \ , \ \ldots, \ p_{m,1}|| \ \ldots || \ p_{m,n_m}] \ \forall p_{j,l} \in \triangle_{p_l}^{C_{in}}$

  if $e_j = p'_j : \beta, \beta = (p_a \ \omega \ p_b)$ // comparing values of paths p_a and p_b

  then

    $e_{j_{ext}} = p_{a,1} \mid \ldots \mid p_{a,n_a} \ \omega \ p_{b,1} \mid \ldots \mid p_{b,n_b}$, $\forall p_{a,k} \in \triangle_{p_a}^{C_{in}}$, $\forall p_{b,l} \in \triangle_{p_b}^{C_{in}}$

  $E_{ext} = E_{ext} \cup \{e_{j_{ext}}\}$ // extending E_{ext} by the new extended element $e_{j_{ext}}$

end

return $\pi_{E_{ext}}$

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

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

The query reformulation engine start first by extracting the following entries from the dictionary:

- the absolute paths leading to the path *year*, i.e., $\triangle_{year}^{C}$, are *[year, info.year, film.details.year, description.year]*
- the absolute paths leading to the path *director.first_name*, i.e., $\triangle_{director.first\_name}^{C}$, are *[director.first_name, info.people.director.first_name, film.details.director.first_name, description.director.first_name]*
- the absolute paths leading to the path *lead_actor.first_name*, i.e., $\triangle_{lead\_actor.first\_name}^{C}$, are *[lead_actor.first_name, info.people.lead_actor.first_name, film.details.personas.lead_actor.first_name, description.stars.lead_actor.first_name]*

- the absolute paths leading to the path *genres*, i.e., $\triangle^{C}_{genres}$, are *[genres, info.genres, film.details.genres, classification.genres]*
- the absolute paths leading to the path *title*, i.e., $\triangle^{C}_{title}$, are *[title, film.title, description.title]*
- the absolute paths leading to the path *ranking.score*, i.e., $\triangle^{C}_{ranking.score}$, are *[ranking.score, info.ranking.score, film.others.ranking.score, classification.ranking.score]*

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

- the element cond:director.first_name = lead_actor.first_name becomes:

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

  where

  p'_a = director.first_name | info.people.director.first_name | film.details.director.first_name | description.director.first_name

  p'_b = lead_actor.first_name | info.people.lead_actor.first_name | film.details.personas.lead_actor.first_name | description.stars.lead_actor.first_name

- the element desc:[title, genres] becomes:

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

  where

  p'_1 =title || film.title || description.title

  p'_2 = genres || info.genres || film.details.genres || classification.genres

- the element production_year:year becomes:

  production_year: year | info.year | film.details.year | description.year

- the element ranking.score becomes:

  ranking.score | info.ranking.score | film.others.ranking.score | classification.ranking.score

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

$$\pi_{\text{cond:director.first\_name | info.people.director.first\_name | film.details.director.first\_name | description.director.first\_name = lead\_actor.first\_name | info.people.lead\_actor.first\_name | film.details.personas.lead\_actor.first\_name | description.stars.lead\_actor.first\_name, desc:[title || film.title || description.title, genres || info.genres || film.details.genres || classification.genres], production\_year:year | info.year | film.details.year | description.year, ranking.score | info.ranking.score | film.others.ranking.score | classification.ranking.score}}(C) \quad (C)$$

The execution of this latest project operator returns:

- {
  "_id":1.0,
  "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 Aggregation

Definition 4.5. Aggregation

The aggregate operator is defined as:

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

The aggregate operator (γ) is a unary operator grouping documents according to the values from the grouping conditions G. The output is a collection of documents where each document refers to one group and contains a computed aggregated value over the group as defined by the aggregation function F (Ben Hamadou et al., 2018b).

- 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 Figure 4.1:

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

4.4.1 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 *iii)* d have all the same values $\forall i\ h_i \in H$;

- 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 such that $p_f \in S_d$, then f is computed across all documents d_i of the group where $p_f \in S_{d_i}$; documents d_k of the group where $p_f \notin S_{d_k}$ are simply ignored,
  - If $\nexists\ d$, a document from the group, such that $p_f \in S_d$, then f is evaluated as a *null* value regardless of its original value.

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_count":3
}
```


-
```
{
 "_id":8.1,
 "titles_count":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 these documents.

4.4.2 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 can 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:

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

In the following we introduce the Algorithm 8 to present the automatic process of reformulating the aggregation operator. Algorithm 8 starts first by generating projection elements from the attributes expressed in the group by and aggregation functions, Lines $2-5$.

Algorithm 8: Algorithm for automatic aggregate operator reformulation.

```
   Input : G γ F
1  E_ext ← ∅
2  foreach p_j ∈ G ∪ {p_f}                    // for each attribute in G and F
3  do
4  |  E_ext = E_ext ∪ {p_j : p_{j,1}|...|p_{j,n_j}}, ∀p_{j,l} ∈ △_{p_j}^{C_in}   // generating elements
   |   where paths △_{p_j}^{C_in} are renamed to p_j
5  end
6  return G γ F ∘ π_{E_ext}
```

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

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

To reformulate the aggregate operator, the query reformulation engine must first generate a project operator using the following dictionary entries:

- the absolute paths leading to the path *title*, i.e., $\triangle_{title}^{C}$, are *[title, film.title, description.title]*

- the absolute paths leading to the path *ranking.score*, i.e., $\triangle^{C}_{ranking.score}$, are *[ranking.score, info.ranking.score, film.others.ranking.score, classification.ranking.score]*

Therefore, it generates the following projection operation:

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

The aggregate operator after reformulation becomes:

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

Now after executing this query we obtain the following results:

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

4.5 Unnest

Definition 4.6. Unnest

The unnest operator is defined as:

$$\mu_{\text{p}}(\text{C}_{\text{in}}) = \text{C}_{\text{out}}$$

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. Let us notice that the output collection C_{out} contains at least equal number of documents as the collection C_{in}, and usually more documents (Ben Hamadou et al., 2019b).

Example. Let us suppose that we want to run the following unnest operation on collection (C) from Figure 4.1:

$$\mu_{\text{genres}}(\text{C})$$

4.5.1 Classical Unnest Evaluation

During an unnest evaluation, classical query engines generate new documents for the operation $\mu_{\mathrm{p}}(\mathrm{C_{in}}) = \mathrm{C_{out}}$ as follows: For a document $d_i \in C_{in}$

- If $p \in S_{d_i}$, and its value is an array $[v_{p_1}, \ldots, v_{p_n}]$ 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 is a copy of d_i and contains the path p. The value of p in each new document $d_{i,j}$ is equal to the j^{th} entry from the array value v_p in d_i;
- If $p \in S_{d_i}$ and its value is atomic or object and not an array, the collection C_{out} contains the same document d_i and the same number of documents as C_{in}.
- If $p \notin S_{d_i}$, the collection C_{out} contains a copy of the original document d_i.

Example. The previous unnest operator considers 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}
  }

Let us notice that documents (b,c,d) are present in the result since they do not include path genres. The result contains five documents.

4.5.2 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:

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

In the following we introduce the Algorithm 9 to present the automatic process of reformulating the unnset operator.

Algorithm 9: Algorithm for automatic unnset operator reformulation.

```
   Input  : μ_p
1  μ_ext ← ∅
2  foreach p_j ∈ △_p^{C_in};          // for each attribute p_j in △_p^{C_in}
3   do
4  |  μ_ext ← μ_ext ∘ μ_{p_j};          // extending μ_ext with μ_{p_j}
5  end
6  return μ_ext
```

Algorithm 9 generates a composition of unnest operators for each absolute paths found in the dictionary leading to attributes from the initial queries. The results are automatically added to the extended query μ_{ext}, Lines $1-4$.

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

$$\mu_{\mathrm{genres}}(\mathrm{C})$$

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

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

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):

```
• {                                          Baby",
   "_id":1,                                "year":2004,
   "title":"Million Dollar                 "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}


       "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"},
  }
  ```

- ```
  {
   "_id":1,
   "title":"Million Dollar Baby",
   "year":2004,
   "genres":"Sport",
   "country":"USA",
   "director":{


    "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}
    }
   }
  }

• {
   "_id":4,
   "description":{
    "title":"The Good, the
          Bad and the Ugly",
    "year":1966,
    "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"
   }
  }
```


4.6 Lookup

Definition 4.7. Lookup

The lookup operator is defined as:

$$(\mathrm{C_{in}})\ \lambda_{\mathrm{res:p_{in}=p_{ex}}}(\mathrm{C_{ex}}) = \mathrm{C_{out}}$$

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 are used to nest or unnest values, it is important to underline that lookup and unnest operators are not reverse operators (Ben Hamadou et al., 2019b).

Example. Let us suppose that we want to run the following lookup operation on collection (C) from Figure 4.1:

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

This 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.

4.6.1 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 execution of the previous 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":[
 {
  "_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}
  }
 ]
```


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":[{
 "_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}
  }
  }
 },
 {
  "_id":4,
  "description":{
  "title":"The Good, the Bad
      and the Ugly",
  "year":1966,
  "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"]
  }
 }
]
```


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.

4.6.2 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:

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

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

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

The query reformulation engine start first by extracting the following entries from the dictionary:

- the absolute paths leading to the path *director.first_name*, i.e., $\triangle^{C}_{director.first\_name}$, are *[director.first_name, info.people.director.first_name, film.details.director.first_name, description.director.first_name]*

- the absolute paths leading to the path *lead_actor.first_name*, i.e., $\triangle^{C}_{lead\_actor.first\_name}$, are *[lead_actor.first_name, info.people.lead_actor.first_name, film.details.personas.lead_actor.first_name, description.stars.lead_actor.first_name]*

Below is the reformulation of the lookup operation:

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

The execution of this lookup operation gives four documents. First, it gives these three 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}
   }
  }
 }
]
```

Second, 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 for document (d):

- `"dir_actor":[]`

4.7 Algorithm for Automatic Query Reformulation

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

If we take into account the definition of a user query (section 4.1), the goal of the extension algorithm 10 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 $q_1 \circ \ldots \circ q_r$.

Algorithm 10 starts by initialising the query Q_{ext} with the identity *id*, line 3. Then, for each operator q_i in the query Q, Lines 4. The algorithm proceeds as follows for each of the five supported operators; *i)* project operator, i.e., π, the algorithm executes the instructions from algorithm 7, Lines $8-30$. *ii)* select operator, i.e., σ, the algorithm executes the reformulation rules defined for this operator, Lines $31-36$. *iii)* aggregate operator, i.e., γ, the Algorithm executes the instructions from algorithm 8, Lines $37-46$. *iv)* unnest operator, i.e., μ, the algorithm executes the instructions from Algorithm 9, Lines $47-53$. *v)* lookup operator, i.e., λ, the algorithm executes the reformulation rules defined for this operator, Lines $55-57$. Finally, the algorithm return the extended query Q_{ext}, Line 60.

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.

4.8 Conclusion

In this chapter, we introduced most document querying operators. First, we started by defining each operator and we presented their classical evaluation. To enable the user to formulate schema-independent querying for heterogeneous document stores where queries could be formulated over partial as well as absolute paths, we introduced for each operator a set of reformulation rules. Our contribution is built upon the idea of using the underlying query engine of the document stores. Therefore, we extend each element from the query with their corresponding absolute paths extracted from a materialised dictionary defined in the previous chapter. We support user queries formulated over partial information regarding the paths leading to the information of

Algorithm 10: Automatic query reformulation algorithm.

input: Q // original query
output: Q_{ext} // reformulated query
$Q_{ext} \leftarrow id$ // identity
foreach $q_i \in Q$ // for each operator in Q
do
  switch q_i // case of the operator q_i
  do
    case π_E : // q_i is a project operator
    do
      $E_{ext} \leftarrow \varnothing$ // initialising the set of extended elements from E
      foreach $e_j \in E$ // for each element $e_j \in E$
      do
        if $e_j = p_j$ *is a path* $(p_j \in P_{C_{in}})$ // e_j takes the form of a path
        then
          $e_{j_{ext}} = p_{j,1} \mid \ldots \mid p_{j,n_{n_j}} \forall p_{j,l} \in \triangle_{p_j}^{C_{in}}$ // generating $p_{j_{ext}}$ using paths from $\triangle_{p_j}^{C_{in}}$
        if $e_j = p'_j : p_j$, $(p_j \in P_{C_{in}})$ // renaming the path p_j to p'_j
        then
          $e_{j_{ext}} = p'_j : p_{j,1} \mid \ldots \mid p_{j,n_j}$, $\forall p_{j,l} \in \triangle_{p_j}^{C_{in}}$ // generating $e_{j_{ext}}$ while renaming paths from $\triangle_{p_j}^{C_{in}}$ to p'_j
        if $e_j = p'_j : [p_1, \ldots, p_{m_j}]$, $\forall l \in [1..m_j], p_l \in S_{C_{in}}$ // new array $[p_1, \ldots, p_{m_j}]$ composed of paths p_l
        then
          $e_{j_{ext}} = p'_j : \left[p_{1,1} || \ \ldots || \ p_{1,n_1} \ , \ \ldots, \ p_{m,1} || \ \ldots || \ p_{m,n_m}\right] \ \forall p_{j,l} \in \triangle_{p_l}^{C_{in}}$
        if $e_j = p'_j : \beta, \beta = (p_a \ \omega \ p_b)$ // comparing values of paths p_a and p_b
        then
          $e_{j_{ext}} = p_{a,1} \mid \ldots \mid p_{a,n_a} \ \omega \ p_{b,1} \mid \ldots \mid p_{b,n_b}$, $\forall p_{a,k} \in \triangle_{p_a}^{C_{in}}$, $\forall p_{b,l} \in \triangle_{p_b}^{C_{in}}$
        $E_{ext} = E_{ext} \cup \{e_{j_{ext}}\}$ // extending E_{ext} by the new extended element $e_{j_{ext}}$
      end
      $Q_{ext} \leftarrow Q_{ext} \circ \pi_{E_{ext}}$ // adding the extended projection $\pi_{E_{ext}}$ to Q_{ext}
    end
    case σ_P : // q_i is a select operator and the condition is normalised to $P = \bigwedge\left(\bigvee p_k \ \omega_k \ v_k\right)$
    do
      $P_{ext} \leftarrow \bigwedge\left(\bigvee(\bigvee_{\forall p_j \in \triangle_{p_k}^{C_{in}}} p_j \ \omega_k \ v_k)\right)$ // extending the condition with a disjunction $\bigvee_{\forall p_j \in \triangle_{p_k}^{C_{in}}} p_j \ \omega_k \ v_k$
      $Q_{ext} \leftarrow Q_{ext} \circ \sigma_{P_{ext}}$ // adding the extended selection $\sigma_{P_{ext}}$ to Q_{ext}
    end
    case ${}_G\gamma_F$:
    where $G = p_1, \ \ldots, \ p_g$, *and* $F = p : f(p_f)$ // q_i is an aggregate operator
    do
      $E_{ext} \leftarrow \varnothing$
      foreach $p_j \in \{G\} \cup \{p_f\}$ // for each attribute in G and F
      do
        $E_{ext} = E_{ext} \cup \{p_j : p_{j,1} | \ldots | p_{j,n_j}\}, \forall p_{j,l} \in \triangle_{p_j}^{C_{in}}$ // $\triangle_{p_j}^{C_{in}}$ are renamed to p_j
      end
      $Q_{ext} \leftarrow Q_{ext} \circ ({}_G\gamma_F \circ \pi_{E_{ext}})$ // adding the combined aggregation ${}_G\gamma_F$ and the custom projection $\pi_{E_{ext}}$ to Q_{ext}
    end
    case μ_p : // q_i is an unnest operation
    do
      foreach $p_j \in \triangle_p^{C_{in}}$ // for each attribute p_j in $\triangle_p^{C_{in}}$
      do
        $Q_{ext} \leftarrow Q_{ext} \circ \mu_{p_j}$ // extending Q_{ext} with μ_{p_j}
      end
    end
    case $\lambda_{res:p_{in}=p_{ex}}$: // q_i is a lookup operation
    do
      $Q_{ext} \leftarrow Q_{ext} \circ \lambda_{res:p_{j,1} \mid \ldots \mid p_{j,n_j} \ = \ p_{l,1} \mid \ldots \mid p_{l,n_l}} \ \forall p_{j,x} \in \triangle_{p_{in}}^{C_{in}}, \ \forall p_{l,y} \in \triangle_{p_{ex}}^{C_{ex}}$
    end
  end
end
return Q_{ext}

interest. Thus, we extend the capabilities of most document stores to support queries that are not formulated using only absolute paths.

In our contribution, the query reformulation Algorithm is designed to support any kind of heterogeneity. Since the dictionary and the Algorithm are independent, it is possible to define a custom dictionary to overcome given classes of heterogeneity, e.g., semantic heterogeneity, thus our Algorithm delivers results that respond to the matching already defined in the dictionary. Furthermore, it is possible to formulate queries using custom attributes names. The core of contribution is flexible and could be adaptable for any class of heterogeneity and queries could be formulated over several types of attribute representations since the latter should be already defined as dictionary entries.

In this thesis, we introduced the different reformulation rules to enable querying for schema-independent querying for heterogeneous collection of documents. Thus, to highlight the importance of introducing such a solution, we focused on overcoming a given type of heterogeneity, i.e., structural heterogeneity. However, all formal definitions that we introduced in this thesis could be easily adapted to support other classes of heterogeneity.

In the next chapter, we introduce the different experiments to validate all the different formal definitions that we introduced in these two previous chapters. Thus, we developed a tool that we called *EasyQ*, that stands for Easy Query, that *i)* extracts document structures, *ii)* maintains the dictionary, and *iii)* automatically reformulates user queries. Later, we introduce our experimental protocol. Finally, we present the results of all the experiments conducted for the aim of this thesis.

Chapter 5

Evaluation

Contents

Along this thesis manuscript, we introduced formal definitions related to the concept of the document model, structures, etc. Then, we introduced the concept of a dictionary to track the different structures of documents within a collection of a heterogeneous collection of documents. Later on, we introduced a set of reformulation rules to enable schema-independent querying for heterogeneous collection of documents using queries formulated over partial or absolute paths. Thus, our contribution involves three main components to *i)* extract document structures, *ii)* maintain the dictionary, and *iii)* automatically reformulate user queries, that we try to validate and evaluate in this chapter. In the following, we present *i)* the implementation of this three components as a prototype that we called *EasyQ*, and *ii)* the experiments that we conducted to evaluate the efficiency of our contribution.

In practical terms, the purpose of the experiments is to answer the following questions:

- is the time to build the dictionary acceptable, and is the size of the dictionary affected by the number of structures in the collection?
- what is the cost of maintaining the dictionary?
- 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?

For this purpose, we start first by describing the synthetic dataset that we employ for both structure extraction and query reformulation evaluation. Later, we start by evaluating the schema inference techniques and the dictionary construction phase using the Algorithm 2, that we introduced in Chapter 3 describing the process of construction a dictionary for a collection of heterogeneous collection of documents. Therein, we evaluate the reformulation of the queries using the Automatic Query Reformulation Algorithm 10, Chapter 3, and we compare the overhead of executing the extended queries to two other execution contexts. Then, we evaluate the dictionary maintenance costs. Our goal is (i) to demonstrate that all formal definitions in this thesis are feasible and reliable, (ii) to analyse the performances of the document structures inference and the query reformulation.

5.1 Implementing EasyQ

5.1.1 Architecture Overview

In this part, we present the architecture of *EasyQ* and its main components.

Figure 5.1 provides a high-level illustration of the architecture of *EasyQ* with its two main components: the query reformulation engine and the dictionary. Moreover,

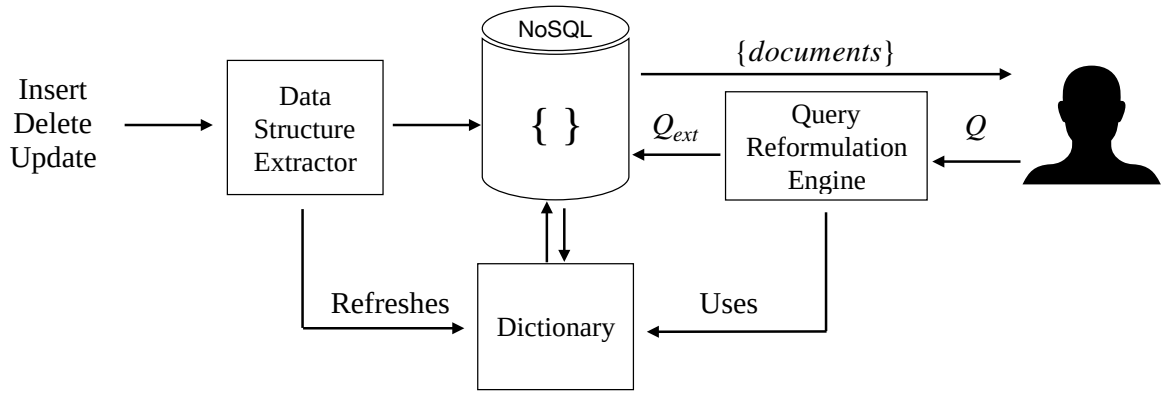

Figure 5.1: EasyQ architecture: data structure extractor and query reformulation engine.

Figure 5.1 shows the flow of data during the data loading stage and the query processing stage.

To evaluate and validate the different formal models introduced in this thesis, we developed *EasyQ* as a proof of concept for all Algorithms introduced in Chapter 3 and Chapter 4. Thus, *EasyQ* helps us to query and manipulate a collection of heterogeneous documents in automatic ways. Hence, *EasyQ* ensure the interaction between the document store and the user. Regarding our technical choices, we implemented *EasyQ* using *Python* programming language. As for the document store, we employed *MongoDB* which is one of the most commonly used document stores. All interactions with the document store were ensured using the library *PyMongo*. The usage of our implementation of *EasyQ* is ensured using the command line. *EasyQ* is mainly composed of two main components described as follows:

- *i)* the data structure extractor: This module ensures that the dictionary is always updated with all existing structures present inside each collection. The data structure extractor module runs a recursive algorithm that goes through all the trees of documents 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. This module offers the following functions:

  - the first execution scenario is automatically executed whenever a document manipulation operation is executed. Therefore, it enriches the dictionary with new partial path entries and updates existing ones with all corresponding absolute paths in documents. Thus, all manipulation operations are intercepted by *EasyQ* and this module automatically detects the changes and generates a new dictionary in case of creating a new collection, or updates existing dictionary in case of executing any manipulation operation. The execution of this scenario is transparent to the user and it is executed as a

background process, and no interaction with the user is required.

- the second execution scenario is manually launched by the user. It allows to generate a dictionary for a given heterogeneous collection of documents. Hence, the user executes a command line on which she specifies the name of the collection. Afterwords, *EasyQ* display to the user in the command line the dictionary and information regarding the time required to generate the dictionary.

- *ii)* the query reformulation engine: At the querying stage, *EasyQ* takes as input the user query, denoted by Q, which is formulated using any combination of paths (leaf nodes, partial paths and absolute paths) and the desired collection. Then, the 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. Later, this module sends this new query to the underlying document store querying engine. Once the query is executed, results are automatically displayed to the user in the command line and information regarding the execution time are displayed.

5.2 Experimental Protocol

In this section, we introduce the different configurations that we employed to evaluate the performances of our contribution. We start first by presenting the experiments environment and the tools used to conduct all experiments. Therein, we describe the synthetic datasets helping to study the heterogeneity effects on the different contribution presented in this thesis. Later, we introduce the different workloads that we employ in these experiments. Finally, we describe the different execution contexts that we run to evaluate the queries.

5.2.1 Experimental Environment

In this part we present the experimental environment that we built to evaluate *EasyQ* and validate its ability for enabling schema-independent querying for NoSQL document-oriented databases and automatically maintaining the dictionary with the latest document structures.

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 16GB 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.

5.2.2 Datasets

In this part we describe the different customised synthetic datasets that we generated to run our experiments. We choose to work with synthetic datasets in order to adjust the different parameters that helps to study the effects of heterogeneity on the execution of the reformulated queries. Hence, we could set the number of nesting levels, the number of attributes within a given complex attribute, the number of documents per collection, etc.

```
{  "_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
   }
}
```

Figure 5.2: Document from the *Baseline* dataset.

In order to analyse the behaviour of *EasyQ* on varying collections sizes and structures, we generated customised synthetic datasets. First, we collected a CSV document with information related to 5,000 movies. We initially extracted all movies details from an online database of information related to films, *IMDB*[1]. 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, e.g., *country*, 3 array type attributes, e.g., *genres*, and 3 complex attributes of an object type, e.g., *ranking* in which we nested additional primitive attributes). All documents within the different collections in the *Baseline* dataset share the same structure as illustrated in Figure 5.2. 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

[1]https://www.imdb.com/

```
{"_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"]
                   }
           }
         }
       }
     }
}
```


Figure 5.3: Document from the *Heterogeneous* dataset (3 groups, 5 nesting levels).

Baseline dataset was composed of five collections of 1M, 10M, 25M, 50M, 100M and 500M documents for a total disk space ranging from 500MB to more than 250GB. In the baseline dataset, there are 1M of distinct documents where attributes values are selected automatically from the initial 5,000 distinct documents used to generate our synthetic datasets. In case of a collection of 25M document, each document is repeated 25 times and so on.

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. The process of generating the heterogeneous collection took several parameters into account: the number of distinct structures, the depth of the absolute paths and the number of new complex attributes in which are nested attributes used in baseline dataset. We randomly nested a subset of attributes, for instance, 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.

Figure 5.3 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 distinct structures. For instance, the number of the new complex attributes in which are nested the attributes from the baseline dataset is equal to three, i.e., *group_1A, group_2A, group_3A*. The depth of the absolute paths in this example is equal to five. The number of nested attributes per complex attribute is equal to five for the first group, one for the second group, and four for the third group.

For the purpose of the experiments we used the above mentioned strategy to generate five datasets described as follows:

- a *Heterogeneous* dataset to evaluate the execution time of the reformulated query on varying collections sizes. This dataset was composed of five collections of 1M, 10M, 25M, 50M, 100M and 500M documents for a total disk space ranging from 500MB to more than 250GB 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 100M documents with *10, 100, 1,000, 3,000 and 5,000* schemas respectively for more than 50GB of disk space for each collection;

- 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 10M documents with *10, 20, 50, 100 and 200* schemas respectively for more than 5GB of disk space for each collection;

- a *Loaded* dataset to evaluate the dictionary construction time on an existing collections. This dataset was composed of five collections of 200M documents containing 2, 4, 6, 8 and 10 schemas respectively for more than 100GB of disk space for each collection;

- an *Adhoc* dataset to evaluate the dictionary construction time for the loading collections phase. This dataset was composed of five collections of 1M of documents containing 2, 4, 6, 8 and 10 schemas respectively for more than 1GB of disk space for each collection.

- a *Manipulation* dataset to evaluate the dictionary update time on executing a manipulation operation. This dataset was composed of five collections of 1k, 10k, 100k, 300k and 500k documents containing 10 schemas for a total disk space ranging from 50MB to more than 2.5GB.

In Table 5.1, we represent the characteristics of documents in the *Heterogenous* dataset. The characteristics in terms of nesting levels and the number of grouping objects is automatically selected for the remaining datasets, i.e., *Schemas* dataset, *Structures*, *Loaded* and *Adhoc* datasets. Therefore, the number of grouping object per schema is between 1 and 7, the number of nesting levels is between 1 and 8.

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 5.1: Settings of the *Heterogeneous* dataset for query reformulation evaluation.

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). Therefore, 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.

5.2.3 Workloads

In this part, we define the different workloads that helps to evaluate the query reformulation engine on varying collections of documents. We built two workloads composed of a synthetic series of queries; *i)* an *operator evaluation* to evaluate separately the execution time of each reformulated operator selection-projection-aggregation-unnest-lookup, 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 nodes, e.g., *score*. The following is the projection query that we used in our experiments:

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

- for the selection operation we chose to build a query that covers the classical comparison operators, i.e., {<,>,≤,≥,=, ≠} for numerical values, e.g., (year ≥ 2004) as well as classical logical operators, i.e., {and:∧, or:∨} between query predicates (e.g., ((year ≥ 2004) ∨ (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:

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

- 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:

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

- 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:

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

- 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 3,033 entries, and we built a lookup query that enriches movie documents with details of the lead actor in each movie. We do not inject any structural heterogeneity to the "actors" collection. The following is the lookup query that we used in our experiments:

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

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":

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

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

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

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

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

Table 5.2 highlights the different characteristics of the selected attributes in queries from both workloads and gives details about their depth inside documents of the *Heterogeneous dataset*.

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}

Table 5.2: Workloads query elements.

In our queries we employed 11 attributes of different types (primitive, e.g., *link*, and complex *genres*) 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 5.3 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 $Q_{4,ext}$.

In Table 5.4, we present a summary of all queries that we employ in our experiments for both workloads.

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

Table 5.3: The number of extracted documents per the two workloads using *Heterogeneous* dataset.

operator evaluation	$Q_1 : \pi_{\text{cond:director.first\_name = lead\_actor.first\_name, desc:[title, genres.1], production\_year:year, score}}(\text{C})$ $Q_2 : \sigma_{(\text{year} \geq 2004 \vee \text{,genres.1 = "Drama"}) \wedge (\text{ranking = \{"score": 6 \} } \vee \text{ link} \neq \text{null})}(C)$ $Q_3 : {}_{\text{country}}\gamma_{\text{maximum\_score:Max(score)}}(\text{C})$ $Q_4 : \mu_{\text{awards}}(\text{C})$ $Q_5 : (\text{C})\lambda_{\text{res:actors.1=actor}}(\text{actors})$ $Q_6 : \pi_{\text{cond:director.first\_name = lead\_actor.first\_name, desc:[title, genres.1], production\_year:year, score}}(\mu_{\text{awards}}(\text{C}))$
operator combination evaluation	$Q_7 : \pi_{\text{cond:director.first\_name = lead\_actor.first\_name, desc:[title, genres.1], production\_year:year, score}}(\sigma_{(\text{year} \geq 2004 \vee \text{genres.1 = Drama}) \wedge (\text{ranking = \{score: 6 \} } \vee \text{ link} \neq \text{null})}(C))$ $Q_8 : \pi_{\text{cond:director.first\_name = lead\_actor.first\_name, desc:[title, genres.1], production\_year:year, score}}(\sigma_{(\text{year} \geq 2004 \vee \text{genres.1 = Drama}) \wedge (\text{ranking = \{"score": 6 \} } \vee \text{ link} \neq \text{null})}(\mu_{\text{awards}}(\text{C})))$

Table 5.4: Summary of the different queries used in the experiments

5.2.4 Execution Contexts

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 for each query Q from the two workloads as follows:

- Q_{Base} is the name of the query that refers to the initial user query (one of the queries from from the two above workloads): 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 reformulated by our approach. It was executed across the *Heterogeneous* dataset;

- $Q_{Accumulated}$ refers to distinct queries where each query Q 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 and the required time is not considered. 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 sequentially.

5.3 Schema Inference Evaluation

In this section, we focus on the dictionary construction process. *EasyQ* offers the possibility of building the dictionary from the scratch for an existing collection. Furthermore, we try to push the dictionary to its limit and to see if the dictionary can handle collections with large number of structures. In these experiments, we study the characteristics of the dictionary, e.g., size and time to construct the dictionary.

5.3.1 Dictionary Construction

In this part, we evaluate the performances of inferring structures of heterogeneous collection of documents. For this purpose, we evaluated the time required to build the dictionary for collections from the *Loaded* dataset on varying number of structures.

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

Table 5.5: Time to build the dictionary for collections from the *Loaded* dataset (100GB, 200M documents).

We can see from the results in Table 5.5 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 5.5 we can see the negligible size of the generated dictionaries when compared to the 100GB of the collection, i.e., around 22KB.

5.3.2 Dictionary at the Scale

In this part, we try to push the dictionary construction phase to its limit. Therefore, we ran experiments to construct dictionary of collection of varying structures, i.e., ranging from 10 schemas to 5,000 schemas using *Schemas* dataset. We notice that our dictionary can support up to 5,000 distinct schemas, which is the limit for the number of schemas we decided on for the purpose of this experiment. We believe that current data-intensive application could not reach such high number of heterogeneous schemas to manage simultaneously. The resulting size of the materialized dictionary is very promising because it does not require significant storage space. In table 5.6 the size of a dictionary for a collection having 5,000 schemas do not exceeds 12MB when compared to the size of the collections around 100GB for a collection having 5,000 schemas.

# of schemas	Dictionary size
10	40KB
100	74KB
1k	2MB
3k	7.2MB
5k	12MB

Table 5.6: Number of schemas effects on dictionary size using *Schemas* dataset.

In this section, we validated the formal definition introduced in Chapter 4 related to the document data model and the dictionary. We demonstrate that the implementation of the Algorithm 2, introduced in Chapter 3 which generates the dictionary, can infer schemas and construct a dictionary for a collection of varying structures at up to 100GB of data. Furthermore, we showed that our Algorithm can infer structures from collections with up to 5,000 distinct schemas. The time to construct the dictionary increases when the number of heterogeneous structures increases within the studied collection. This behavior is because the Algorithm 2 considers each document individually to infer its structure whereas state-of-the-art solution elects a subset of documents from which they infer their structures. For instance, Apache Spark (Zaharia et al., 2016) do not infer structures from all documents thus leading to incoherent query results due to lack of full information regarding all structures.

5.4 Queries Evaluation Results

In this section, we discuss the performances of executing the reformulated queries on varying collections of heterogeneous schemas. Furthermore, we evaluate the performances of reformulating queries using the Algorithm 10 introduced in Chapter 4 for ensuring the automatic query reformulation.

5.4.1 Reformulated Queries Performances

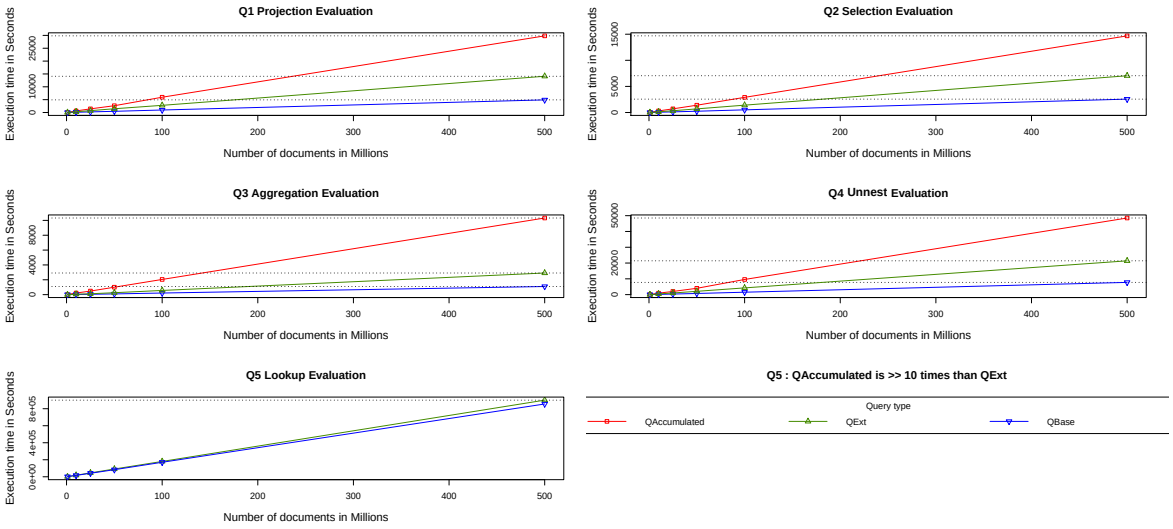

Figure 5.4: *operator evaluation* workload using *heterogeneous* dataset.

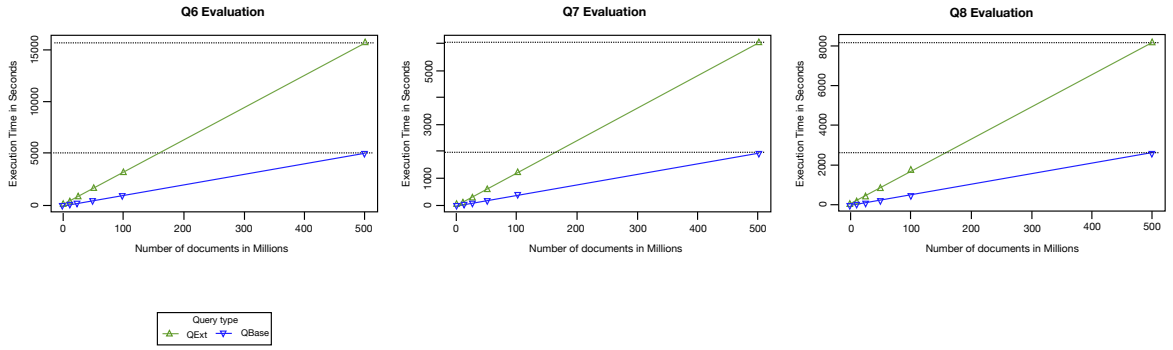

Figure 5.5: *operator combination evaluation* workload using *heterogeneous* dataset.

In this part, all queries are executed using the *heterogeneous* dataset composed of five collections of 1M, 10M, 25M, 50M, 100M and 500M documents for a total disk space ranging from 500MB to more than 250GB and each collection contained 10 schemas.

As shown in Figure 5.4, 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 reformulation engine 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 examine now 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 Figure 5.5, 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 share the same behaviour as the overhead arising from the execution of the single execution of each operator, it is not exponential, (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 execution of the queries over the baseline dataset occurs when studying the effects of querying varying number of documents. 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 present the time needed to execute the reformulated query in Table 5.7. 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 comparisons required to execute each query. For instance, the execution of the query Q_6 over the collection having 200 schemas requires 200 possible paths for each attribute of the six attributes involved in the query. In sum, the extension of the query Q_6 contains 1,2000 absolute paths.

# of Schemas	10	20	50	100	200
Time in (s)	200	380	690	1,140	2,560

Table 5.7: Evaluating Q_6 on varying number of schemas, *Structures* dataset.

In these experiments, we push each parameter to its limit like the number of schemas, size of the collection, ... to study the robustness of our system. The results show that EasyQ scales well and offer acceptable rates.

5.4.2 Query Reformulation Time

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 5.8. It is notable that the time to generate the reformulated query is less than two seconds, which is very low when to compared to the time and efforts to build similar query manually. In this series of experiments, we have tried to find distinct navigational paths for seven predicates. Each rewritten query is composed of numerous absolute paths for each predicate. Table 5.8 shows that reformulation of the query Q_6 initially formulated using seven paths generates queries having 70 absolute paths for a collection of 10 schemas, 700 absolute paths for a collection of 100 schemas, 7,000 absolute paths for a collection of 1,000 schemas, 21,000 absolute paths for a collection of 3,000 schemas and 35,000 absolute paths for a collection of 5,000 schemas. We believe that the query rewriting engine scales 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 35,000 comparisons per document when dealing with a collection containing 5,000 distinct schemas.

In this section, we demonstrate the efficiency of our reformulation rules to retrieve relevant queries and that those queries could be executed using the native mechanisms of most underlying document stores. All queries generated using the implementation of the Algorithm 10, introduced in Chapter 5 to automatically reformulating the queries,

# of schemas	# of absolute paths	Reformulation time
10	70	0.0005s
100	700	0.0025s
1k	7k	0.139s
3k	21k	0.6s
5k	35k	1.52s

Table 5.8: Number of schema effects on query rewriting (# of paths in reformulated query and reformulation time) (query Q_6) over *Schemas* dataset.

overcome the structural heterogeneity within documents and deliver expected results. We notice that the time required to execute the reformulated queries is relatively higher than the normal execution of a query having unique schema. We explain this overhead by the additional evaluations required to be performed by the query engine to find out the adequate query parameters for each structure of documents. The main advantage of our query reformulation is that we offer users the possibility to run queries using partial information regarding the paths whereas most document stores require full path to retrieve information. Furthermore, we demonstrated that we are able with one query to retrieve data from a collection having 5k schemas. The query reformulation takes only 1.2s which is very promising.

5.5 Dictionary Maintenance Evaluation

In this section, we study the dictionary maintenance process. The data structure extractor module in *EasyQ* offers the possibility to build the dictionary for existing dataset. Furthermore, it offers the possibility to maintain existing dictionaries to keep them with the latest version of the data each time the collection of documents is manipulated. Hence, the query reformulation engine uses the latest structures found in the different collections. However, if the process of data manipulation is in progress, it may not take into consideration the recent changes. In the following, we study for each manipulation operation, i.e., inset, delete and update, the performances of maintaining the dictionary with information about the latest absolute paths and possible paths to use while formulating queries. First, we start by the evaluation of the time required to build the dictionary using *Heterogeneous* dataset while inserting data. Later, we evaluate the cost of updating the dictionary while deleting documents from a collection of documents. Finally, we evaluate the cost of updating documents. Both delete and update operation are evaluated using *Manipulation* dataset. For all experiments, we measure the time required to execute a manipulation operation without updating the dictionary, we refer to this evaluation as the baseline since we employ native mechanisms of MongoDB. Later, we measure the time required to execute the manipulation operation while updating the dictionary. Finally, we show the overhead induced by

our system.

5.5.1 Dictionary Update on Insert Operation

In this experiment, we evaluate the performances of the Algorithm 4, introduced in Chapter 4, to update the dictionary during an insert manipulation operation using the *Adhoc* dataset. We notice from the results in the Table 5.9 that the time elapsed to refresh the dictionary increases when we start to deal with collections having more heterogeneity. In case of the collection with 10 structures, the time does not exceed 40% when we compare it to a collection with 2 structures.

#of schema	MongoDB	EasyQ	Overhead
2	201s	269s	33%
4	205s	277s	35%
6	207s	285s	37%
8	208s	300s	44%
10	210s	309s	47%

Table 5.9: Manipulation evaluation: insert operation using *Manipulation* dataset.

Table 5.9 shows that for 1 M of documents and for collections of up to 10 distinct schema the overhead does not exceed 47%. We find that the overhead measure does not exceed 0.5 the time required to load data on MongoDB. The evolution of the time when compared to the number of number of schemas in the collection is linear and is not exponential which is encouraging.

5.5.2 Dictionary Update on Delete Operation

In this experiment, we evaluate the performances of the Algorithm 5, introduced in Chapter 4, for updating the dictionary during a delete manipulation operation using the *manipulation* dataset. We notice from the results in the Table 5.10 that the time elapsed to refresh the dictionary increases when we start to deal with collections having more documents. Furthermore, we notice that the overhead added to delete and refresh the dictionary while deleting a set of 500k documents does not exceed 1.4 seconds which is just 18% of overhead.

We notice from Table 5.10 for the delete operation that the overhead added to the execution of the delete operation is similar to the execution of an insert operation, e.g., does not exceed 48%.

5.5.3 Dictionary Update on Documents Update Operation

In this experiment, we evaluate the performances of the Algorithm 6 updating the dictionary during an update manipulation operation using the *manipulation* dataset.

#of documents	MongoDB	EasyQ	Overhead
1k	0.03s	0.04s	33%
10k	0.15s	0.204s	36%
100k	0.8s	1.112s	39%
300k	2.2s	3.146s	43%
500k	3s	4.44s	48%

Table 5.10: Manipulation evaluation: delete operation using *Manipulation* dataset.

We notice from the results in the Table 5.11 that the time elapsed to build the dictionary increases when we start to deal with collections having more heterogeneity. In case of the collection with 500k documents, the overhead of executing the update manipulation and updating the dictionary exceeds 120%. However, this overhead that may reach 1.2 times because we do not employ optimisation for this operator in this thesis. Furthermore, executing an update operation requires to run a delete and an insert manipulation operation over both the database and the dictionary. Thus, each operation requires to build temporary dictionaries etc. We estimate that operation of updating structures of documents are frequent as update operation to update attributes values. Thus, in this thesis we do not propose any sophisticated optimisation to accelerate this process. We address this issue in our future work.

#of Documents	MongoDB	EasyQ	Overhead
1k	1.3s	2s	53%
10k	14s	24s	71%
100k	149s	285s	91%
300k	183s	380s	107%
500k	239s	527s	120%

Table 5.11: Manipulation evaluation: update operation using *Manipulation* dataset.

In this section, we validated the dictionary maintenance process by implementing the Algorithms [4,5,6]. Results show that refreshing the dictionary requires additional time when compared to the normal execution of each manipulation operation over collection of documents. We studied this time and we discovered that it does not exceed 48% for all of the insert and delete operators. However, we found that the time required to maintain the dictionary requires up to 1.2 times the time required to execute the update operation only. The main advantage of our approach is that the dictionary is updated on the fly. Thus, it is not necessary to infer structures from documents which are already stored within a collection. Therefore, all maintenance Algorithms work to infer structures for only affected documents by these operations. Maintaining the dictionary and refreshing it is beneficial to overcome the structural heterogeneity within collection of heterogeneous documents. This process ensures that

all queries are reformulated on executing time with a guarantee that the query contains only valid absolute paths from the refreshed dictionary.

5.6 Conclusion

In this chapter, we validated the different formal definitions that we introduced for the document data model from Chapter 3 and we validate the Algorithm 10 for automatic query reformulation from Chapter 4. We proved that our solution to capture heterogeneity within a collection of heterogeneous documents could handle collections of large volume of data, i.e., up to 500M documents in a collection of 250GB size. Furthermore, we pushed the dictionary to its limitations in terms of maximum number of heterogeneous schemas and results show that we could construct a dictionary for a collection having 5,000 distinct schemas.

The validation of the dictionary construction phase and its capacity to handle large volume of data and high number of structures within the same collection encourage us to run a second series of experimentations to evaluate the main contribution of this thesis introduced in Chapter 5 consisting of reformulating initial users queries with the usage of the dictionary. 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 thesis is higher when compared to the execution of basic user queries, but always less than a multiple of three. Nevertheless, this time overhead seems to be acceptable when compared to the execution and the merge of results of separated queries built manually for each schema while heterogeneity issues are automatically managed. Furthermore, we succeeded to reformulate and execute queries over a collection having 5k schemas. Results are very promising since the time required to reformulate queries on such heterogeneous collection of 5k schemas does not exceeds 1.5 second.

In order to maintain the correctness of the query and to optimise the query reformulation by excluding obsolete absolute paths, we introduced a set of automatic mechanisms to refresh the dictionary each time a manipulation execution is launched. Results shows that the overhead added in this process is acceptable in both insert and delete manipulation operation. However, update operator requires additional tuning and optimisation which is the subject of our future work.

All the implementation and the experiments introduced in this Chapter helped us to validate the different contribution of this thesis. The main purpose was to prove the feasibility and the validity of all formal definition introduced in this thesis. Optimisation and real uses cases are under study and are a good subject for our

future work. Currently, we are deploying EasyQ as a mainstream solution to query sensors data in the aim of the neOCampus [2] project at the campus of the University of Toulouse III-Paul Sabatier. In this project we are gathering sensors data, the structure of the data is not unique and thus we are experimenting EasyQ to enable different collaborators to access heterogeneous sensors data through a dedicated web API.

[2]https://www.irit.fr/neocampus

Chapter 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 thesis 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 consists in generating and maintaining a 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 in transparent ways for users. The query reformulation can be applied to most document store operators based on formal foundations that are stated in the thesis.

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 thesis 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 thesis, 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 thesis 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.

Current extensions of this work consists of adopting the kernel of operators to other data models. In our recent work published in (El Malki et al., 2018), we succeeded to overcome the heterogeneity in graphs and we provided support for further class of heterogeneity, i.e., semantic and syntactic. However, we supported only a subset of operators and we working on extending the support to cover all operators introduced in this thesis. Another interesting extension of the present work is the usage of the query reformulation rules in a polystore systems. In a another recent joint work, we published the paper (Ben Hamadou et al., 2019a), where we employed the query reformulation rules introduced in this thesis and we showed that they were useful to be adopted for overcoming further class of heterogeneity. In that work, heterogeneities cover both the data model and the structures.

Future research work will cover the different aspects presented in this thesis. 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. Thus, we are experimenting *EasyQ* in the context of neOCampus project at the University of Toulouse-III Paul Sabatier. For the query reformulation process we will enable support for more document operations, e.g., join. 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.

Bibliography

D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

S. Amer-Yahia, F. Du, and J. Freire. A comprehensive solution to the xml-to-relational mapping problem. In *Proceedings of the 6th annual ACM international workshop on Web information and data management*, pages 31–38. ACM, 2004.

J. C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: The Definitive Guide: Time to Relax*. " O'Reilly Media, Inc.", 2010.

M. Armbrust, R. S. Xin, C. Lian, Y. Huai, and Liu. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD*, pages 1383–1394. ACM, 2015.

M.-A. Baazizi, H. B. Lahmar, D. Colazzo, G. Ghelli, and C. Sartiani. Schema inference for massive json datasets. In *(EDBT)*, pages 222–233, 2017.

K. Banker. *MongoDB in action*. Manning Publications Co., 2011.

H. Ben Hamadou, F. Ghozzi, A. Péninou, and 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.*, 2018a.

H. Ben Hamadou, F. Ghozzi, A. Péninou, and 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, Volume 1.*, pages 58–68, 2018b.

H. Ben Hamadou, F. Ghozzi, A. Péninou, and O. Teste. Interrogation de données structurellement hétérogènes dans les bases de données orientées documents (regular paper). In *Journées Francophones Extraction et Gestion de Connaissances (EGC 2018), Paris, 22/01/2018-26/01/2018*, volume vol.RNTI-E-34, pages 155–166. Revue des Nouvelles Technologies de l'Information (RNTI), janvier 2018c.

H. Ben Hamadou, E. Gallinucci, and M. Golfarelli. Answering GPSJ Queries in a Polystore: a Dataspace-Based Approach (regular paper). In *International Conference on Conceptual Modeling (ER 2019), salvador,Brazil, 04/11/2019*, https://link.springer.com, novembre 2019a. Springer.

H. Ben Hamadou, F. Ghozzi, A. Péninou, and O. Teste. Schema-independent querying for heterogeneous collections in nosql document stores. *Information Systems*, 85:48–67, 2019b. URL `https://doi.org/10.1016/j.is.2019.04.005`.

H. Ben Hamadou, F. Ghozzi, A. Péninou, and O. Teste. Schema-Independent Querying and Manipulation for Heterogeneous Collections in NoSQL Document Stores. In *Enterprise Information Systems*, volume 363 of *LNBIP*, chapter 16, pages 1–26. Springer Nature, 2019c.

E. Bertino, G. Guerrini, and M. Mesiti. A matching algorithm for measuring the structural similarity between an xml document and a dtd and its applications. *Information Systems*, 29(1):23–46, 2004.

P. Bille. A survey on tree edit distance and related problems. *Theoretical computer science*, 337(1-3):217–239, 2005.

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

T. Böhme and E. Rahm. Supporting efficient streaming and insertion of xml data in rdbms. In *DIWeb*, pages 70–81, 2004.

E. Botoeva, D. Calvanese, B. Cogrel, and G. Xiao. Expressivity and complexity of mongodb queries. In *21st International Conference on Database Theory, ICDT 2018, March 26-29, 2018, Vienna, Austria*, pages 9:1–9:23, 2018.

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

J. L. Carlson. *Redis in action*. Manning Publications Co., 2013.

R. Cattell. Scalable sql and nosql data stores. *Acm Sigmod Record*, 39(4):12–27, 2011.

Š. Čebirić, F. Goasdoué, and I. Manolescu. Query-oriented summarization of rdf graphs. *Proceedings of the VLDB Endowment*, 8(12):2012–2015, 2015.

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

M. Chavalier, M. El Malki, A. Kopliku, O. Teste, and R. Tournier. Document-oriented data warehouses: Models and extended cuboids, extended cuboids in oriented document. In *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*, pages 1–11. IEEE, 2016.

C. P. Chen and C.-Y. Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information sciences*, 275:314–347, 2014.

M. Chevalier, M. El Malki, A. Kopliku, O. Teste, and R. Tournier. Implementation of multidimensional databases in column-oriented nosql systems. In *East European Conference on Advances in Databases and Information Systems*, pages 79–91. Springer, 2015.

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

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

W. W. W. Consortium et al. Extensible markup language (xml) 1.1. 2006.

A. Corbellini, C. Mateos, A. Zunino, D. Godoy, and S. Schiaffino. Persisting big-data: The nosql landscape. *Information Systems*, 63:1–23, 2017.

M. DiScala and 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*, pages 295–310. ACM, 2016.

H.-H. Do and E. Rahm. Coma system for flexible combination of schema matching approaches. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 610–621. Elsevier, 2002.

M. El Malki. *Modélisation NoSQL des entrepôts de données multidimensionnelles massives*. Thèse de doctorat, Université de Toulouse-le-Mirail, Toulouse, France, décembre 2016.

M. El Malki, H. Ben Hamadou, M. Chevalier, A. Péninou, and O. Teste. Querying heterogeneous data in graph-oriented NoSQL systems (short paper). In C. Ordonez and L. Bellatreche, editors, *International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2018), Regensburg, Germany, 03/09/2018-06/09/2018*, September 2018. ISBN 978-3-319-98539-8.

A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang. Can the elephants handle the nosql onslaught? *Proceedings of the VLDB Endowment*, 5(12):1712–1723, 2012.

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

D. Florescu and D. Kossmann. Storing and querying xml data using an rdmbs. *IEEE data engineering bulletin*, 22:3, 1999.

N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1433–1445. ACM, 2018.

D. D. Freydenberger and T. Kötzing. Fast learning of restricted regular expressions and dtds. *Theory of Computing Systems*, 57(4):1114–1158, 2015.

E. Gallinucci, M. Golfarelli, and S. Rizzi. Schema profiling of document-oriented databases. *Information Systems*, 75:13–25, 2018.

P. Gómez, R. Casallas, and C. Roncancio. Data schema does matter, even in nosql systems! In *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*, pages 1–6. IEEE, 2016.

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

P. A. Hall and G. R. Dowling. Approximate string matching. *ACM computing surveys (CSUR)*, 12(4):381–402, 1980.

M. Hausenblas and J. Nadeau. Apache drill: interactive ad-hoc analysis at scale. *Big Data*, 1(2):100–104, 2013.

R. Hecht and S. Jablonski. Nosql evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 336–341. IEEE, 2011.

J. Hidders, J. Paredaens, and 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*, pages 137–149. ACM, 2017.

S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. Monetdb: Two decades of research in column-oriented database. 2012.

A. A. Imam, S. Basri, R. Ahmad, J. Watada, and M. T. González-Aparicio. Automatic schema suggestion model for nosql document-stores databases. *Journal of Big Data*, 5(1):46, 2018.

C. G. C. Index. Forecast and methodology, 2014–2019, 2015. *Forrás: www. cisco. com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper. pdf (2015. 10. 01.)*.

J. Kang and J. F. Naughton. On schema matching with opaque column names and data values. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 205–216. ACM, 2003.

K. Kaur and R. Rani. Modeling and querying data in nosql databases. In *2013 IEEE International Conference on Big Data*, pages 1–7. IEEE, 2013.

H. F. Korth and M. A. Roth. Query languages for nested relational databases. In *Workshop on Theory and Applications of Nested Relations and Complex Objects*, pages 190–204. Springer, 1987.

A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

M. Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246. ACM, 2002.

X. Li, S. Szpakowicz, and S. Matwin. A wordnet-based algorithm for word sense disambiguation. In *IJCAI*, volume 95, pages 1368–1374, 1995.

L. Lim, H. Wang, and M. Wang. Semantic queries in databases: problems and challenges. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 1505–1508. ACM, 2009.

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

L. Lin, V. Lychagina, W. Liu, Y. Kwon, S. Mittal, and M. Wong. Tenzing a sql implementation on the mapreduce framework. 2011.

J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with cupid. In *vldb*, volume 1, pages 49–58, 2001.

J. Madhavan, P. A. Bernstein, A. Doan, and A. Halevy. Corpus-based schema matching. In *21st International Conference on Data Engineering (ICDE'05)*, pages 57–68. IEEE, 2005.

A. McAfee, E. Brynjolfsson, T. H. Davenport, D. Patil, and D. Barton. Big data: the management revolution. *Harvard business review*, 90(10):60–68, 2012.

S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

B. Momjian. *PostgreSQL: introduction and concepts*, volume 192. Addison-Wesley New York, 2001.

J. Murty. *Programming amazon web services: S3, EC2, SQS, FPS, and SimpleDB.* " O'Reilly Media, Inc.", 2008.

A. Nayak, A. Poriya, and D. Poojary. Type of nosql databases and its comparison with relational databases. *International Journal of Applied Information Systems*, 5 (4):16–19, 2013.

N. H. O'Donnell. Storied lives on instagram: Factors associated with the need for personal-visual identity. *Visual Communication Quarterly*, 25(3):131–142, 2018.

K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The sql++ query language: Configurable, unifying and semi-structured. *arXiv preprint arXiv:1405.3631*, 2014.

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

F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273. International World Wide Web Conferences Steering Committee, 2016.

J. Pokorny. Nosql databases: a step to database scalability in web environment. *International Journal of Web Information Systems*, 9(1):69–82, 2013.

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

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

R. W. Schvaneveldt. *Pathfinder associative networks: Studies in knowledge organization.* Ablex Publishing, 1990.

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

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

M. Stonebraker. New opportunities for new sql. *Communications of the ACM*, 5(11): 10–11, 2012.

M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

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

V. Ventrone. Semantic heterogeneity as a result of domain evolution. *ACM SIGMOD Record*, 20(4):16–20, 1991.

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*, pages 171–180. ACM, 1993.

A. Vukotic, N. Watt, T. Abedrabbo, D. Fox, and J. Partner. *Neo4j in action*, volume 22. Manning Shelter Island, 2015.

J. Wang, J.-R. Wen, F. Lochovsky, and 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*, pages 408–419. VLDB Endowment, 2004.

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

D. B. West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, NJ, 1996.

A. Wolski. Linda: A system for loosely integrated databases. In *[1989] Proceedings. Fifth International Conference on Data Engineering*, pages 66–73. IEEE, 1989.

M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

P. Zikopoulos, C. Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.