EXODuS: Exploratory OLAP over Document Stores[☆]

Mohamed L. Chouder[a], Stefano Rizzi[b], Rachid Chalal[a]

[a]*LMCS, ESI, Algiers, Algeria*
[b]*DISI — CINI, University of Bologna, Bologna, Italy*

Abstract

OLAP has been extensively used for a couple of decades as a data analysis approach to support decision making on enterprise structured data. Now, with the wide diffusion of NoSQL databases holding semi-structured data, there is a growing need for enabling OLAP on document stores as well, to allow non-expert users to get new insights and make better decisions. Unfortunately, due to their schemaless nature, document stores are hardly accessible via direct OLAP querying. In this paper we propose EXODuS, an interactive, schema-on-read approach to enable OLAP querying of document stores in the context of self-service BI and exploratory OLAP. To discover multidimensional hierarchies in document stores we adopt a data-driven approach based on the mining of approximate functional dependencies; to ensure good performances, we incrementally build local portions of hierarchies for the levels involved in the current user query. Users execute an analysis session by expressing well-formed multidimensional queries related by OLAP operations; these queries are then translated into the native query language of MongoDB, one of the most popular document-based DBMS. An experimental evaluation on real-world datasets shows the efficiency of our approach and its compatibility with a real-time setting.

Keywords: Document stores, JSON, Exploratory OLAP, Self-service BI, Multidimensional modeling

[☆]This work was partly supported by the EU-funded project TOREADOR (contract n. H2020-688797).

1. Introduction

Over the past decade, companies have been adopting NoSQL databases to deal with the huge volumes of data manipulated by modern applications. NoSQL systems have emerged as an alternative to relational database management systems in several implementations [1]. They can be classified based on their data model, the most popular categories being key-value, wide-column, graph-based and document-oriented. In particular, document-oriented databases (briefly, *document stores*) are considered to be very developer-friendly, thus they have attracted a large interest from researchers and practitioners; indeed, they offer a flexible data model with great query possibilities, are very easy to maintain, and offer rich APIs [2].

Document stores collect nested, denormalized, and hierarchical documents. Documents are self-describing and mainly encoded using the semi-structured data format JSON (JavaScript Object Notation). Documents are organized in *collections*; in compliance with the *data first, schema later or never* paradigm, the documents within the same collection may present a structural variety to ensure flexibility and support evolution. This *schemaless* nature provides a "fluid" data model that has attracted developers seeking to avoid the restrictions posed by the relational model.

The growing use of document stores and the dominance of JSON have resulted in vast amounts of semi-structured data holding precious information, which could be profitably integrated into existing business intelligence (BI) systems [3]. On-Line Analytical Processing (OLAP) is the querying paradigm normally used in the context of BI to analyze multidimensional (MD) data stored in data warehouses and data cubes, and it also has been recognized to be an effective way for conducting analytics over big NoSQL data as well [4]. Unfortunately, due to their schemaless nature, document stores are hardly accessible via direct OLAP querying. Recent efforts in this area propose to enable SQL querying of schemaless data for analytic purposes, since SQL is well supported by several BI tools [5, 6, 7]. However, none of these solutions provides OLAP on document stores.

In principle, two main approaches can be followed for enabling OLAP over a data source: schema-on-write and schema-on-read. A *schema-on-write* approach would force a fixed MD structure in data and load them into a data warehouse to be then queried, while a *schema-on-read* approach would leave data unchanged in their structure until they are accessed by the user [8]. Classical MD design follows a schema-on-write approach, in that it requires a target MD schema to be designed and an ETL (Extract, Transform, and Load) process to be set before posing any queries. Conversely, in this paper we follow a schema-on-read approach, which we claim should be preferred when querying document stores in an OLAP fashion for the following reasons:

- In a schema-on-read approach, the schema can be defined at query time based on user queries, which allows data to be directly analyzed without requiring an upfront investment in schema and ETL design [7].
- A schema-on-read approach does not necessarily require an intervention of IT people, so it is more suitable to *self-service BI* scenarios —where the search and integration of data is accomplished by users without any mediation by analysts, designers, or programmers [3].
- A schema-on-read approach can operate in real-time, so it is suitable to *exploratory OLAP* scenarios [9] —where data scientists and data enthusiasts need to timely access *situational data*, i.e., data with a narrow focus on a specific domain problem and a short lifespan [10].
- Document stores handle high volumes of data with varied structure, which question the ability of traditional ETL in processing data on the one hand, and the possibility of storing them into a data warehouse with a fixed schema on the other.
- Due to the evolving nature of schemaless data, ETL and schema maintenance in a schema-on-write approach would become more complex than for enterprise structured data.

In this paper we propose EXODuS, an interactive, schema-on-read approach to enable OLAP querying of document stores in the context of self-service BI and exploratory OLAP. OLAP requires data to be in MD form, i.e., organized into measures, dimensions with different levels of aggregation (MD *hierarchies*). To discover hierarchies despite the lack of structure, we adopt a data-driven approach based on the mining of *approximate functional dependencies* (AFDs); to ensure good performances, we incrementally build local portions of hierarchies for the levels involved in the current user query, thus acting in an on-demand fashion. EXODuS enables users to execute an analysis session by expressing well-formed MD queries related by OLAP operations. These queries are then translated into the native query language of MongoDB, one of the most popular document-based

DBMS (`www.db-engines.com/en/ranking`), taking advantage of the discovered MD knowledge. One significant benefit of our approach is that it allows non-expert users to analyze data in document stores by means of operations they already know, such as roll-up and drill-down, without having a deep understanding of how data is actually stored. This allows them to create reports in real-time and in a self-service fashion, which reduces the time and effort spent in traditional BI settings.

The rest of the paper is organized as follows. Section 2 presents an overview of EXODuS and Section 3 discusses the related work. In Section 4 the three phases of our approach are detailed. Section 5 presents the experimental evaluation and Section 6 concludes the paper giving some future directions for research.

2. Preliminaries

In this section we give the basics of document stores followed by a high-level overview of the three steps of our approach: MD Enrichment, Querying, and OLAP enabling.

2.1. Document Stores

A document store holds a set of databases, each organizing the storage of documents in the form of *collections*. A *document*, also called *object*, consists of a set of name-value pairs (names are also called *keys*) typically encoded using JSON (`www.json.org`). JSON is a popular data exchange format, widely adopted in modern web applications (a formal specification of JSON is proposed in [11]). It has many variants that are used for storage and optimization purposes, such as BSON (`www.bsonspec.org`) in MongoDB and, very recently, OSON in the Oracle DBMS [6]. Keys in a JSON document are always strings, while values usually have the following types: *primitive* (number, string, date, Boolean), *object*, and *array* of primitive values or objects. Therefore, a document can contain nested, denormalized, and hierarchical data.

From a conceptual point of view, a typical collection consists of a set of business objects connected through two kinds of relationships: nesting and references. Nesting consists of embedding objects arbitrarily within other objects, which corresponds to two types of relationships: to-one in the case of a nested object and to-many in the case of an array of nested objects. On

```
order :
{ “order_id” : 100,
  “date” : “2017-03-27”,
  “customer_id” : 1005,
  “items” :
  [ { “product_name” : “Keyboard”,
      “quantity” : 5,
      “unitprice” : 20,
      “rating” : 5
    },
    { “product_name” : “Webcam”,
      “quantity” : 1,
      “unitprice” : 35,
      “rating” : 4
    }
  ]
  “totalprice” : 135
}
```

(a)

```
order:                              item :
{ “order_id” : 100,                 { “order_id” : 100,
  “date” : “2017-03-27”,              “product_id” : 85,
  “customer_id” : 1005,               “quantity” : 1,
  “totalprice” : 135                  “unitprice” : 35,
}                                     “rating” : 4
                                    }
                                    item :
                                    { “order_id” : 100,
                                      “product_id” : 86,
                                      “quantity” : 5,
                                      “unitprice” : 20,
                                      “rating” : 5
                                    }
```

(b)

Figure 1: Sample JSON document of an order: denormalized and nested (a) and normalized (b)

the other hand, references are suitable for modeling many-to-many relationships similarly to foreign keys in relational databases; they can be expressed manually or using a specific mechanism such as `$ref` in MongoDB. However, when references are used, getting data requires joins which are not supported in document stores; so, nesting is most often used instead. A sample JSON document of an order is illustrated in Figure 1 in two schema variants: one denormalized and nested, and one normalized.

In this work we focus on collections of denormalized and nested JSON documents that logically represent the same business entities, while expecting that their structure may vary. Since the approach is data-driven, it also works when nested structures are not present within the input data (e.g., for flat documents), which is relevant because a large number of non-nested datasets are available in JSON format on the web (e.g., more than 11000 collections in `www.data.gov`).

2.2. Approach Overview

With reference to the example in Figure 1, we propose a basic use case. A non-expert user wants to compute some summaries on the orders collection, such as counting orders per customer. She lacks knowledge about JSON and the document store native query language, but she is familiar with traditional BI tools, so she knows how to (i) formulate MD queries by just picking

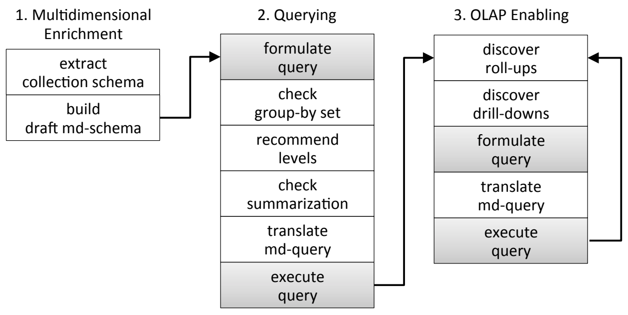

Figure 2: Approach overview; steps in grey are up to either the user of the document store

levels and measures of interest, and (ii) perform OLAP operations by navigating through hierarchies. In a traditional BI setting these queries would be formulated on data with a fixed MD schema whose correctness is ensured by the designer. In a self-service BI or exploratory OLAP setting the user has to determine a proper MD schema on-the-fly by identifying dimensions, measures with their granularities, and hierarchies. To achieve this goal, EXODuS operates in three phases as outlined below and sketched in Figure 2:

1. **MD enrichment** aims to extract a draft MD schema on which the user will be able to write her first query. Since documents in a collection are self-describing, they are not accompanied with a schema definition. Therefore, in this step, a *collection schema* that captures attributes (i.e., paths among document keys), their types, and some relationships between these attributes is extracted from the collection. Then, a draft MD schema is built by classifying attributes into levels and measures based on their type (for example, a numeric attribute is potentially a measure). To provide correct summarization of data, measures are then related to the subset of levels that determine their granularity by checking that the basic MD constraints are met.

2. **Querying**. Now the user can formulate her first MD query by picking levels and measures of interest from the current MD schema; this query is checked by mining AFDs from data to ensure its MD validity and correct summarization. To avoid the exponential complexity of checking all AFDs, we adopt a smart exploration strategy that reduces the

search space by considering level cardinalities and structural information provided by the collection schema. If an AFD is found to hold between a pair of levels, the hierarchies are refined accordingly and some possible querying alternatives are proposed to the user. If the query is found to be well-formed it is translated into the native query language of the document store and executed, and the next phase is triggered.

3. **OLAP enabling** is an iterative phase that enables the user to further explore data by running an OLAP session. To this end, local portions of hierarchies (consisting of roll-up and drill-down relationships for each level involved in the user query) are incrementally built by mining AFDs from data. The user can now apply an OLAP operator (roll-up, drill-down) to iteratively create a new query on the collection, which again is translated into the query language of the document store and executed.

These phases are described in detail in Section 4 together with their automated steps (those represented in white in Figure 2).

3. Related Work

Supply-driven MD design. The automation of MD modeling has been widely explored in the area of data warehouse design. Supply-driven approaches automate the discovery of MD concepts by a thorough analysis of the source data. The first approaches proposed algorithms to build MD schemata starting from Entity/Relationship diagrams or relational schemata [12, 13, 14], while further approaches considered more expressive conceptual schemata such as UML class diagrams [15] and ontologies [16]. In particular, similarly to the querying phase of our approach, [17] proposes an algorithm to check the MD validity of an SQL cube query and to derive the underlying MD schema. This algorithm identifies MD concepts from the structure of the query itself and by following foreign keys in the relational schema.

In these approaches, MD modeling is done at design time, mainly based on FDs expressed in the source schemata as foreign keys or many-to-one relationships. Conversely, our approach is meant to be used at query time and automates the discovery of MD concepts by mining FDs from data, as required by the schemaless context.

MD design from non-relational data. Our approach closely relates to previous approaches for MD modeling from semi-structured data [18, 19, 20] in XML format, which is similar to JSON. These approaches take in input DTDs or XML schemata that provide rich information about XML documents (e.g., multiplicities, data types), so they cannot operate directly on XML data not having a schema specification. In particular, the work in [20] builds MD schemata starting from an XML schema but, in some cases, data is accessed to discover FDs that are not expressed in the schema. Similarly, *starry vault* [21] is a recent approach that mines FDs for MD modeling from data vaults. These are databases characterized by a specific data model tailored to provide historical storage in presence of schema evolutions. The main idea is to mine approximate and temporal FDs to cope with the issue of noisy and time-varying data, which may result in hidden FDs.

All the above-mentioned approaches are similar in that they define MD schemata at design time using structural and additional information extracted from data (i.e., FDs). In contrast, our approach operates at query time and mostly relies on data distributions, while structural information is only used —when available— to reduce the search space.

While several approaches were delivered to implement MD schemata using NoSQL databases, to the best of our knowledge only a very few papers deal with using NoSQL databases as a source for MD design and querying. In particular, in [22] OLAP cubes are built starting from a columnar data warehouse using the MC-CUBE operator, while in [23] the Graph Cube model enables OLAP queries on graph databases. Interestingly, the main methodologies for data warehouse design are compared in [24] at the light of the new requirements posed by big data sources.

A preliminary version of our approach has been proposed in [25]; the new contributions we give in this paper can be summarized as follows:

1. We propose a lightweight algorithm for extracting the schema of a JSON collection.

2. The approach has been extended to cope with the structural variety of JSON collections, assessed through some ad-hoc metrics.

3. The translation of MD queries into the native query language of MongoDB is discussed.

4. A wider set of tests is proposed to evaluate the efficiency and scalability of the approach.

OLAP on linked data. Recent works in this space propose to directly perform OLAP-like analysis over semantic web data. *Exploratory OLAP* has been defined as the process that discovers, collects, integrates, and analyzes these external data on-the-fly [26]. Some works in this direction address the problem of building MD hierarchies from linked data [9, 27], which is closely related to the third phase of our approach. Specifically, *iMOLD* [9] is an instance-based approach that operates at query time for exploratory OLAP on linked data; it aims at finding MD patterns representing roll-up relationships, in order to enable users to extend corporate data cubes with new hierarchies extracted from linked data. Similarly, [27] proposes an algorithm for discovering hierarchies from dimensions present in statistical linked data represented using the RDF Data Cube (QB) vocabulary (`www.w3.org/TR/vocab-data-cube/`). Like starry vault, this algorithm mines for AFDs (here called *quasi*-FDs) to cope with the presence of imperfect and partial data.

The algorithm that we propose for building hierarchies differs from these works in that, to ensure good performances, instead of trying to extensively detect all hierarchies it only looks for *local portions* of hierarchies for the levels involved in the user queries, thus acting in an on-demand fashion.

SQL on schemaless data. Several solutions have emerged in the industry to enable SQL querying of schemaless data for analytic purposes. These solutions can be classified into two categories: (1) relational systems enabling the storage and management of schemaless data, and (2) systems designed as an SQL interface for schemaless data.

Solutions in the first category include RDBMSs that support storage and querying of schemaless data to be used along with relational data in one system [5, 6] (e.g., Oracle, SQL Server, IBM DB2, and PostgreSQL). Current systems do not impose a fixed relational schema to store and query data, but derive and maintain a dynamic schema to be used for schema-on-read SQL/JSON querying instead [6]. Other solutions that fall into this category are virtual adapters that expose a relational view of the data in a document store, to be used in common BI tools; an example is the MongoDB BI connector (`www.mongodb.com/products/bi-connector`).

The second line of solutions are SQL engines that offer extensions to query schemaless data persisted in document stores or as JSON files (e.g., in Hadoop). Spark SQL [28] has been designed for relational data processing of native Spark distributed datasets in addition of diverse data sources and formats (including JSON). In order to run SQL queries on schemaless data in Spark SQL, a schema is automatically inferred beforehand. Apache Drill (`drill.apache.org`) is a distributed SQL engine that can join data from multiple data sources, including Hadoop and NoSQL databases. It has a built-in support for JSON with columnar in-memory execution. Drill is also able to dynamically discover a schema during query processing, which allows to handle schemaless data without defining a schema upfront. Finally, Presto (`prestodb.io`) is a distributed SQL engine, developed at Facebook for interactive queries, that can also combine data from multiple kinds of data sources including relational and non-relational databases. When querying schemaless data, Presto tries to discover data types, but sometimes it may need a schema to be defined manually in order to run queries. All above-mentioned engines differ in their architecture, support to ANSI SQL, and extensions to SQL to handle schemaless and nested data. These engines can be used to query data stored in MongoDB, which gave us various architectural options for implementation. However, none of them supports OLAP on document stores.

Schema discovery from document stores. Several works have addressed schema discovery from document stores [29, 30, 31, 32]. These works focus on the structural variety of documents within the same collection caused by the schemaless nature and by the evolution of data; for instance, in [31] all the distinct schemata in a collection are extracted and stored in a repository. In order to have a single view of a collection, the authors propose a relaxed form of schema called *skeleton* that better captures the core and prominent attributes and filters out unfrequent ones. However, the computational cost for building skeletons is high, making them unsuitable in real-time contexts [31]. The algorithm proposed in [32] extracts a schema for each document and then merges these schemas into a global schema that captures all possible attributes. This algorithm provides better execution times even with massive data sets.

The algorithm that we propose differs from the existing ones in that it pushes all the computation down to the document store, resulting in better execution times. It also captures the structural variety within a collection by computing the occurrence frequencies of the attributes and their value types, without a significant impact on performance.

```
games :
{ "_id" : ObjectId("52f2a70eddbd75540aba7c06"),
  "date" : ISODate("1989-03-27T02:00:00Z"),
  "teams" :
  [ { "name" : "Detroit Pistons",
      "abbreviation" : "DET",
      "score" : 90,
      "home" : true,
      "city" : "Detroit",
      "won" : 1,
      "results" : { "assists" : 20, . . . },
      "players" :
      [ { "player" : "Isiah Thomas",
          "points" : 30, . . . },
        . . .
      ]
    },
    { "name": "Dallas Mavericks",
      "abbreviation" : "DAL",
      "score" : 77,
      "home" : false,
      "city" : "Dallas",
      "won" : 0,
      "results" : { "assists" : 16, . . . },
      "players" :
      [ { "player" : "Rolando Blackman",
          "points" : 23, . . . },
        . . .
      ]
    }
  ]
}
```

(a)

```
games :
{ "_id" : ObjectId("52f29f91ddbd75540aba6dae"),
  "team" :
  { "name" : "Washington Bullets",
    "abbreviation" : "WSB",
    "city" : "Washington"},
  "plays" :
  [ { "id" : 198510254,
      "date" : ISODate("1985-10-25T04:00:00Z"),
      "score" : 100,
      "home" : false,
      "won" : 1,
      "results" : { "assists" : 21, . . . },
      "players" :
      [ { "player" : "Jeff Ruland",
          "points" : 19, . . . },
        . . .
      ]
    },
    { "id" : 198510314,
      "date" : ISODate("1985-10-31T04:00:00Z"),
      "score" : 107,
      "home" : true,
      "won" : 0,
      "results" : { "assists" : 17, . . . },
      "players" :
      [ { "player" : "Dan Roundfield",
          "points" : 12, . . . },
        . . .
      ]
    }, . . .
  ]
}
```

(b)

Figure 3: Sample JSON documents of a game

4. The EXODuS Approach

In this section we give a detailed description of our approach with reference to a real-world example, namely, a collection that models NBA games (adapted from [33]) where a document represents one game between two teams along with players and team results.

Example 1. *In the sample document shown in Figure 3.a the main object is* games*, which embeds multiple* teams*, each including one object* results *and an array of* players*. Since a document store has a flexible schema there are a variety of ways for representing data; clearly, the way data are modeled may affect performance and querying patterns. Figure 3.b shows an alternative JSON representation of the NBA games domain; each document embeds a* team *object and multiple* plays*, each including one object* results *and an array* *of* players*. Note that, in this representation, when the number of games for a team increases, so does the* plays *array; this may lead to reaching the document size limit, in which case data must be split into multiple documents for the same* team*.*

4.1. MD Enrichment

The goal of this phase is to extract a draft MD schema on which the user will be able to formulate her first query. To this end, we extract the schema of the input collection and enrich it with basic MD knowledge as explained in the following subsections.

4.1.1. Extract the Collection Schema

In compliance with the real-time requirement posed by exploratory OLAP, to extract the schema of a collection we propose a lightweight algorithm that accesses the document store and assesses schema variety using COUNT DISTINCT queries. In the scope of this work we only consider the schema variety arising from keys with varying types and missing keys; the stronger variety arising from relevant changes in the document structure (e.g., due to major evolutions and versioning) is not considered. The output of the algorithm is a tree-like schema that provides a single view of the collection and includes all the keys appearing in the documents with their corresponding types, defined as follows.

Definition 1 (Collection Schema). *Given a collection D, its schema (briefly,* c-schema*) is a tree $\mathcal{S} = (K, E)$ where:*

- *K is the union of the sets of keys appearing in the documents of D;*
- *$r \in K$ is the root and is given the collection name;*
- *Each key $k \in K$ has a set of types, denoted $Types(k)$, which can contain simple types (*number*,* string*,* date*,* boolean*,* missing*, and* null*) as well as complex types (*object*, generic* array*,* array of object*, and* array of simple*).*
- *E is a set of arcs that includes (i) an arc $\langle r, k\rangle$ for each key inside the root of the documents in D, and (ii) an arc $\langle k_1, k_2\rangle$ iff k_2 is a key within an object or an array of objects having key k_1.*

Algorithm 1 Extract C-Schema

```
Require: A collection $D$
Ensure: A c-schema $\mathcal{S}$
1: $r \leftarrow nameOf(D)$
2: $K \leftarrow \{r*\}$
3: $E \leftarrow \varnothing$
4: $\mathcal{S} \leftarrow (K, E)$
5: $\mathcal{S} \leftarrow Expand(\mathcal{S}, \{r*\})$
6: return $\mathcal{S}$
```

All keys of array type, including the root, are starred (*) in the c-schema to emphasize that they model -to-many associations.

The pseudo-code for building the c-shema $\mathcal{S}$ is sketched in Algorithms 1 and 2. Algorithm 1 initializes $\mathcal{S}$ with the root key, then calls function *Expand* to build $\mathcal{S}$. The goal of $Expand(\mathcal{S}, R)$ (Algorithm 2) is to extend $\mathcal{S}$ by adding the keys nested in each key of R; each key of type object or array of object is further expanded by recursively invoking the function. At first, for each key $r \in R$, function $RetrieveNestedKeys$ creates and executes on D a query that finds the set of keys K_r nested in r (Line3). The form of this query depends on whether r is an object or an array of object: in the first case, it just returns the nested keys; in the second it flattens the array, limits the type to object, then proceeds as in the first case. Then, for each key $k \in K_r$, if k is an object it is added to the set of keys R' to be expanded in the next recursive call (lines 5—6). If k is an array, a new query is issued to refine its type (e.g., from array to array of simple) and update $Types(k)$ accordingly (lines 7—9). After this update, if k is an array of object, a new key $k*$ (as mentioned above, the * is added because k is an array) is added to $\mathcal{S}$ and to R', with type array of object, to be further expanded (lines 10—14). If k has other types besides array of object, then the latter is removed to let only simple types (lines 15—16); in this case k is added to $\mathcal{S}$ with an arc from r to k (lines 18—19). If k has no other types, it is not added to $\mathcal{S}$ since $k*$ took its place (Line17). Finally, function *Expand* is recursively invoked for the set of keys R' (Line21).

Example 2. *With reference to the document in Figure 3.a, belonging to the* games *collection, we show how the c-schema is built. The first call to Expand is made with the root* games*, of type* object*. The query generated in Line3 first projects each document into an array of objects in the form* [{"k": "name", "v" : value}]*, where k is the key and v its value. Then this array is flattened and the type t of each value v is projected. Now, the documents are*

Algorithm 2 Expand

```
Require: A c-schema $\mathcal{S}$, a set of keys to be expanded $R$
Ensure: An (extended) c-schema $\mathcal{S}$
 1: $R' \leftarrow \varnothing$
 2: for all $r \in R$ do
 3:     $K_r \leftarrow RetrieveNestedKeys(D, r)$
 4:     for all $k \in K_r$ do
 5:         if object $\in Types(k)$ then
 6:             $R' \leftarrow R' \cup \{k\}$
 7:         if array $\in Types(k)$ then
 8:             $T \leftarrow QueryArray(D, k)$
 9:             Update $Types(k)$ based on $T$
10:             if array of object $\in Types(k)$ then
11:                 $R' \leftarrow R' \cup \{k*\}$
12:                 $Types(k*) \leftarrow$ {array of object}
13:                 $K \leftarrow K \cup \{k*\}$
14:                 $E \leftarrow E \cup \{\langle r, k*\rangle\}$
15:                 if $|Types(k)| > 1$ then
16:                     $Types(k) \leftarrow Types(k) \backslash$ {array of object}
17:                 else continue
18:         $K \leftarrow K \cup \{k\}$
19:         $E \leftarrow E \cup \{\langle r, k\rangle\}$
20: if $R' \neq \varnothing$ then
21:     $\mathcal{S} \leftarrow Expand(\mathcal{S}, R')$
22: return $\mathcal{S}$
```

grouped by k and t to return a set of distinct (k, t) pairs while counting their occurrences. Finally, the results are grouped by k as follows:

```
{"k":"_id", "types" : [{"t":"objectId","occ":31686}],
 "k":"teams", "types" : [{"t":"array","occ":31686}],
 "k":"date", "types" : [ {"t":"date","occ":31000},{"t":"string","occ":"563"}{"t":"null","occ":"123"}]}
```

By iterating on each key, teams *is found to be an* array*, so its types are refined by the query in Line8 which flattens the array, projects the value types, and groups by type counting the number of occurrences, to obtain* { "t":"object", "occ" : 31686}*. The type of* teams *is updated to* $Types$(teams) = {array of object} *and a new key* teams* *is added to the c-schema. In the resulting c-schema after the first recursion, the root has children* id, date*, and* teams* *(see Figure 4). In the second call to Expand,* teams* *is expanded to obtain the keys shown in Figure 4 at the third level of the tree. In the third call,* results *(of type* object*) and* players* *(of type* array of object*) are expanded. Then no more objects are found and the algorithm stops. The final c-schema is shown in Figure 4. Similarly, Figure 5 shows the c-schema for the document in Figure 3.b.*

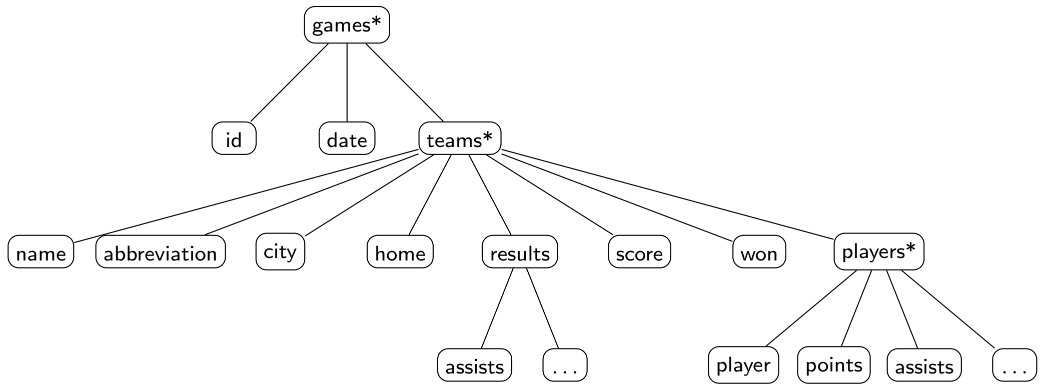

Figure 4: C-schema of the games collection (representation in Figure 3.a)

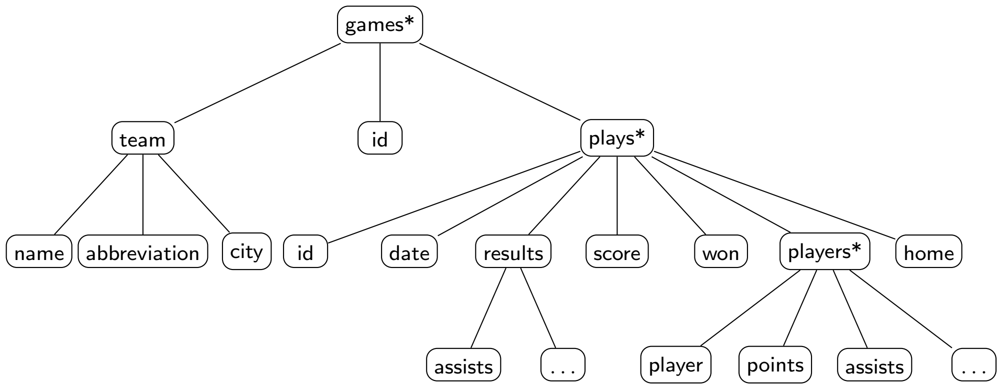

Figure 5: C-schema of the games collection (representation in Figure 3.b)

A path from the root to a leaf of $\mathcal{S}$ is called an *attribute*. To name attributes we use the dot notation omitting the root; so, for instance, the c-schema in Figure 4 contains attributes date, teams*.name, teams*.results.assists, teams*.players*.player, etc. The depth of attribute a is the number of starred keys in its path (including the root), and is denoted by $depth(a)$ (e.g., $depth(\text{date}) = 1$, $depth(\text{teams*.name}) = 2$, $depth(\text{teams*.results.assists}) = 2$); the set of attributes at depth δ is denoted by $Attrs(\delta)$, the maximum depth of the c-schema by δ_{max}. For example, for the c-schema in Figure 4 it is $\delta_{max} = 3$.

Since documents within the same collection may present a structural variety due to schema flexibility and data evolution, they can have varying attributes and an attribute can have different value types. To assess this variability inside a collection we use two metrics. The frequency of attribute a is defined as

$$freq(a) = \frac{|\tilde{k}|}{|D|} \tag{1}$$

where $|D|$ is the total number of documents in the collection, $\tilde{k}$ is the leaf key of a, and $|\tilde{k}|$ is the number of documents where $\tilde{k}$ appears with type different from null and missing. Let $t \in Types(\tilde{k})$ be one of the types of $\tilde{k}$; then, the frequency of type t in a is

$$freq(t, a) = \frac{|t|_{\tilde{k}}}{|\tilde{k}|} \tag{2}$$

where $|t|_{\tilde{k}}$ is the number of documents where $\tilde{k}$ has type t.

4.1.2. Build the Draft Md-Schema

After the c-schema $\mathcal{S}$ of a collection has been defined, a corresponding MD schema has to be derived from it. Since measures at different granularities can possibly be included in the documents (e.g., in our example, points at the player's level and score at the team level), the definition we provide relates each single measure to a specific set of dimensions.

Definition 2 (Md-Schema). *A multidimensional schema (briefly,* md-schema*) is a triple $\mathcal{M} = (H, M, g)$ where:*

- *H is a finite set of* hierarchies*; each hierarchy $h_i \in H$ is associated to a set L_i of categorical* levels *and a* roll-up *partial order $\succeq_i$ of L_i. The top level of each hierarchy is called a* dimension. *Each element $G \in 2^L$, where $L = \bigcup_i L_i$, such that for no $l, l' \in G$ it is $l \succeq_i l'$, is called a* group-by set *of H.*

- *M is a finite set of numerical* measures.

- *g is a function relating each measure in M to the set of levels that determine its* granularity*: $g : M \to \mathcal{G}$ where $\mathcal{G}$ is the set of all group-by sets of H.*

At first, a draft md-schema $\mathcal{M}_{draft}$ is built from $\mathcal{S}$ by tentatively labelling all attributes of types date, string, and Boolean as levels, and all attributes having type number and array of number as measures. In case of an attribute with varying type, labelling is done according to its prevalent type (determined based on Formula 2). The user can contribute to this step by manually changing the label of some attributes, since in some cases a numeric attribute can be used as a level, and a non-numeric attribute can be used as a measure. The first situation has no impact on the following steps, since a group-by set can also include numeric attributes. Conversely, in the second situation, to enable correct aggregations the values of the non-numeric attribute should be transformed into numerical values (in MongoDB, aggregating an attribute with a varying type only takes into account numeric values and skips non-numeric, missing, and null values).

In real collections there can be a notable amount of noise in the form of low-frequency attributes; so, we do not consider the attributes whose frequency (determined based on Formula 1) is below a user-defined threshold. Of course, pruning these attributes may lead to missing some level/measure for analysis. However, including in a query an attribute that is present in a very small percentage of documents would produce very partial results, which could be misleading to the user. Besides, repairing these issues would require specific business rules and cannot be realistically done on-the-fly by an end-user. We claim that some imprecision/incompleteness is a necessary price to be paid to preserve the self-service and real-time nature of our schema-on-read approach.

Since at this stage no roll-up relationship has been discovered yet, each level is the only member of a different hierarchy (i.e., it is a dimension); hence, each possible subset of L is a group-by set. The only exceptions are date dimensions, which can be decomposed into different categorical levels to give rise to standard temporal hierarchies (e.g., date $\geq$ month $\geq$ year). No further attempt to discover hierarchies is made at this stage, since extensively looking for FDs among *all* attributes would be computationally very expensive. As a consequence, in $\mathcal{M}_{draft}$ two levels l and l' might be erroneously modeled as two separate dimensions, while they should be actually part of the same hierarchy because $l \geq l'$. This issue is fixed during the querying and OLAP enabling phases.

To complete the definition of $\mathcal{M}_{draft}$, the granularity mapping g must be built. We recall that an md-schema should comply with the *MD space arrangement constraint*, stating that each instance of a measure is related to one instance of each dimension [17]. In the c-schema $\mathcal{S}$, attributes in $Attrs(\delta)$ are related by to-many multiplicity to those in $Attrs(\delta + 1)$, so a measure cannot be related to levels having a higher depth; specifically, our goal is to avoid that for some $m \in M$ there is a to-many relationship between m and a level $l \in G$, because this would lead to double counting. Therefore, $g(m)$ is set by connecting each measure m to the group-by set G that contains all levels $l \in L$ such that $depth(m) \geqslant depth(l)$. Note that a measure with type array of number can be safely aggregated since a dimension can be related to multiple instances of a measure.

Example 3. *With reference to the c-schema in Figure 4, attribute* teams*.score *is numerical, so it is assumed to be a measure, while* teams*.name *is assumed to be a dimension. It is* $depth(\textsf{teams*.score}) = 2$*, therefore* teams*.score *is tentatively associated in the draft md-schema with group-by set* $G = \{\textsf{teams*.name}, \textsf{teams*.abbreviation}, \textsf{teams*.city}, \textsf{teams*.home}, \textsf{id}, \textsf{date}\}$*. All the measures at the same depth of* teams*.score *(e.g.,* teams*.won *and* teams*.results.assists*) are also associated with* G*. Similarly, measure* teams*.players*.points *has depth 3, so it is associated with group-by set* $G' = \{\textsf{teams*.players*.player}, \textsf{teams*.name}, \textsf{teams*.abbreviation}, \textsf{teams*.city}, \textsf{teams*.home}, \textsf{id}, \textsf{date}\}$*. Formally:*

$$\begin{aligned}
H = {} & \{\{\textsf{teams*.name}\}, \{\textsf{teams*.abbreviation}\}, \\
& \{\textsf{teams*.city}\}, \{\textsf{teams*.home}\}, \{\textsf{id}\}, \{\textsf{date}\}, \\
& \{\textsf{teams*.players*.player}\}\} \\
M = {} & \{\textsf{teams*.score}, \textsf{teams*.won}, \textsf{teams*.results.assists}, \\
& \textsf{teams*.players*.points}\} \\
g(\textsf{teams*.score}) = {} & \{\textsf{teams*.name}, \textsf{teams*.abbreviation}, \textsf{teams*.city}, \\
& \textsf{teams*.home}, \textsf{id}, \textsf{date}\} \\
g(\textsf{teams*.players*.points}) = {} & \{\textsf{teams*.players*.player}, \textsf{teams*.name}, \\
& \textsf{teams*.abbreviation}, \textsf{teams*.city}, \textsf{teams*.home}, \\
& \textsf{id}, \textsf{date}\}
\end{aligned}$$

The draft md-schema is shown in DFM notation [12] in Figure 6; remarkably, it is the same obtained for the c-schema in Figure 5.

The mapping g is defined using information provided by the c-schema, and some exceptions may arise when the relationship between a measure m

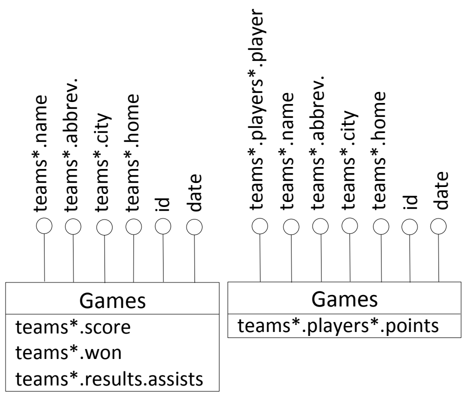

Figure 6: The draft md-schema for games (dimensions are shown as circles, measures are listed within each box)

and a level l cannot be captured. This happens when this relationship is hidden in data, i.e., though m and l have the same depth they are actually related by a to-many relationship [34] (e.g., when arrays of objects are not used to model to-many relationships). In this case, m may be non-additive or even non-aggregable on $g(m)$, so the user's knowledge of the application domain is required to manually fix summarization (see Section 4.2.3).

4.2. Querying

This phase is aimed at supporting a non-expert user in formulating a well-formed MD query. In a schema-on-write approach, queries are formulated on a complete MD schema whose correctness is ensured by the designer. Conversely, in a schema-on-read approach —our case—, the MD schema is defined at read-time by the query itself. Though this approach gives more flexibility because each user can "force" her own MD view onto the data, it requires a further check to ensure that the FDs implied by the query are not contradicted by data; so, querying becomes an iterative process where the underlying md-schema is progressively refined together with the query.

Definition 3 (Md-Query). *A multidimensional query (briefly,* md-query*) q on md-schema $\mathcal{M} = (H, M, g)$ is a triple $q = (G_q, M_q, \Sigma)$ where* [1]

[1]We do not consider selection predicates in this definition because our OLAP enabling phase is focused on roll-ups and drill-downs, which operate on the query group-by set

- *$G_q \in \mathcal{G}$ is the md-query group-by set;*

- *$M_q \subseteq M$ is the set of required measures;*

- *Σ is a function that associates each measure $m \in M_q$ with an aggregation operator.*

Starting from the draft md-schema $\mathcal{M}$, the user formulates an md-query q by choosing one to three dimensions[2] (G_q), one or more measures of interest (M_q), and an aggregation operator for each measure (Σ). However, to be considered well-formed, q must satisfy some conditions.

Firstly, we have to ensure that all the measures in M_q can be correctly aggregated at group-by set G_q. To this end, we observe that the roll-up partial orders on the hierarchies H of an md-schema $\mathcal{M}$ induce a partial order $\succeq_H$ on the set $\mathcal{G}$ of the group-by sets of H, such that $G \succeq_H G'$ iff the granularity of G' is coarser than the one of G [35]. Since in a draft md-schema all hierarchies include a single level with no roll-up relationship, it is $G \succeq_H G'$ when $G' \subseteq G$. Measure $m \in M_q$ is compatible with G_q iff $g(m) \succeq_H G_q$; indeed, if this condition is satisfied, the granularity expressed by the group-by set is coarser than the one at which m is defined, so m can be safely aggregated at G_q.

Example 4. *A possible md-query on the draft md-schema described in Example 3 is the one characterized by $G_q = \{\text{teams*.players*.player}, \text{date}\}$, $M_q = \{\text{teams*.score}\}$, and $\Sigma = \text{Sum}$. It is $g(\text{teams*.score}) \not\succeq_H G_q$ (because* teams*.players*.player *is not in $g(\text{teams*.score})$). Indeed, aggregating on* teams*.score *would result in double counting since each instance of* teams*.score *is related to multiple instances of* teams*.players*.player *(a team has several players).*

Secondly, q (and the md-schema q is formulated on) should comply with the following constraints [17]:

and not on selections. Differently from group-by sets, selections are not influenced by hierarchies, so they would not require additional well-formedness checks.

[2]Group-by sets with more than three dimensions are seldom used in practice, mainly because visualizing n-dimensional datasets with $n \geqslant 4$ requires sophisticated chart types that are typically oriented to very skilled users.

Algorithm 3 Querying

```
Require: A c-schema $\mathcal{S}$, an md-schema $\mathcal{M}$, an md-query $q = (G_q, M_q, \Sigma)$ on $\mathcal{M}$
Ensure: A (refined) md-schema $\mathcal{M}$, a (valid) md-query $q$ on $\mathcal{M}$
1: if $|G_q| > 1$ then
2:     $G_q \leftarrow CheckGroupBy(M_q, G_q, \mathcal{M}, \mathcal{S})$
3:     if $|G_q| = 1$ then
4:         $L_{rec} \leftarrow RecommendLevels(G_q, \mathcal{M})$
5:         Update $G_q$ and $\mathcal{M}$ based on the user's choice
6: $CheckSummarization(q, \mathcal{M})$
7: $TranslateMdQuery(q)$
```

♯1 The *base integrity constraint*, stating that the levels in the group-by set are orthogonal, i.e., functionally independent on each other.

♯2 The *summarization integrity constraint*, which requires *disjointness* (the measure instances to be aggregated are partitioned by the group-by instances), *completeness* (the union of these partitions constitutes the entire set), and *compatibility* (the aggregation operator chosen for each measure is compatible with the type of that measure) [36].

How to carry out these two checks is discussed in the following subsections.

The pseudo-code of the querying phase is sketched in Algorithm 3. It starts from md-query q and md-schema $\mathcal{M}$, and produces in output a valid md-query based on a possibly refined md-schema. Algorithm 3 works as follows. Firstly, if the md-query group-by set includes either 2 or 3 levels (Lines 1–5), procedure *CheckGroupBy* is called to drop from G_q the levels, if any, that are not compliant with the base integrity constraint and update $\mathcal{M}$ accordingly (Line 2). If just one level is left in G_q, procedure *RecommendLevels* is called to look for additional group-by levels (Line 4) and possibly include them in the md-query upon the user's decision (Line 5). Finally, procedure *CheckSummarization* checks summarization (Line 6) and q is translated into the native query language of the document store and executed (Line 7).

4.2.1. Check Group-by Set

This process is aimed at ensuring base integrity and at checking disjointness and completeness [36].

Base integrity requires that the levels in G_q are mutually orthogonal. So, from this point of view, md-query q is valid only if there are no FDs between the levels in G_q, i.e., there are many-to-many relationships between them. An FD is a to-one relationship, usually denoted with $l \rightarrow l'$ to emphasize that values of l functionally determine the values of l'. FDs are not modeled at the schema level, so we must resort to data; since document stores host large amounts of data, we can reasonably assume that the FDs we find are representative enough of the application domain. Unfortunately, schemaless data commonly present errors and missing values, which may hide some FDs. The tool we adopt to cope with this issue are *approximate functional dependencies* (AFDs) [37], which "almost hold" on data; for complexity reasons, we only consider simple AFDs (i.e., those relating single attributes rather than attribute sets), which are mostly common in MD schemata.

Definition 4 (Approximate Functional Dependency). *Given two levels l and l', let $strength(l, l')$ denote the ratio between the number of unique values of l and the number of unique values of ll' [38]. We will say that AFD $l \rightsquigarrow l'$ holds if $strength(l, l') \geqslant \epsilon$, where ϵ is a user-defined threshold.*

To verify base integrity we check AFDs between the levels in G_q; the algorithm we adopt to look for AFDs is *Cords* [39]. Noticeably, to reduce the search complexity we avoid useless AFD checks. An AFD $l \rightsquigarrow l'$ cannot hold —so its check can be avoided— in two cases:

- If the cardinality of l' is higher than the cardinality of l, $|l'| > |l|$.
- If $depth(l) < depth(l')$. Indeed, as already stated in Section 4.1.2, the attributes in $Attrs(\delta)$ are related by to-many multiplicity to those in $Attrs(\delta + 1)$, so $l \not\rightsquigarrow l'$.

The result of all AFD checks performed is stored in a meta-data repository, to be used to avoid checking the same AFDs twice.

The detection of an AFD leads to dropping one or two levels from G_q and to refining the md-schema by creating portions of hierarchies. In particular, when AFD $l \rightsquigarrow l'$ is detected, where $l, l' \in G_q$ and l and l' belong to hierarchies h and h', respectively, l' is dropped from G_q and $\mathcal{M}$ is refined as follows: (i) level l' is moved from h' to h; (ii) the roll-up partial order of h is updated by adding $l \succeq l'$; (iii) the granularity function of each measure m such that $l, l' \in g(m)$ is updated by removing l'; and (iv) h' (which remains empty) is deleted from H.

In more detail, recalling that $1 \leqslant |G_q| \leqslant 3$, the following situations may arise:

1. If $|G_q| = 1$, orthogonality is obvious.

2. If $|G_q| = 2$, the relationship between the two levels in G_q, l and l', is checked. If it turns out to be many-to-many because no AFDs are found between l and l' (e.g., if $G_q = \{\mathsf{date}, \mathsf{teams*.name}\}$), the base integrity constraint is met and both levels remain in G_q. If it turns out to be many-to-one, for instance because $l \rightsquigarrow l'$, then only the level at the "many" side, l, remains in G_q, and $l \geq l'$ is added to $\mathcal{M}$. For instance, if $G_q = \{\mathsf{teams*.name}, \mathsf{teams*.city}\}$, it is $\mathsf{teams*.name} \rightsquigarrow \mathsf{teams*.city}$ so $\mathsf{teams*.city}$ is dropped from G_q and $\mathsf{teams*.name} \geq \mathsf{teams*.city}$ is added to the md-schema.

3. If $|G_q| = 3$, with $G_q = \{l, l', l''\}$, there are four possibilities:

   (a) No AFDs are detected, so many-to-many relationships hold between each pair of levels. In this case the base integrity constraint is met and all levels remain in G_q (e.g., if $G_q = \{\mathsf{date}, \mathsf{teams*.name}, \mathsf{teams*.players*.player}\}$).

   (b) One AFD is detected, say $l' \rightsquigarrow l''$ (e.g., if $G_q = \{\mathsf{teams*.name}, \mathsf{id}, \mathsf{date}\}$, since $\mathsf{id} \rightsquigarrow \mathsf{date}$). In this case, $l' \geq l''$ is added to $\mathcal{M}$ and l'' is removed from G_q.

   (c) Two AFDs are detected. Two cases arise here:

      i. If $l \rightsquigarrow l'$ and $l \rightsquigarrow l''$, both $l \geq l'$ and $l \geq l''$ are added to $\mathcal{M}$, and l' and l'' are dropped from G_q. Conceptually, l is the starting level of a branch in the hierarchy [12].

      ii. If $l' \rightsquigarrow l$ and $l'' \rightsquigarrow l$, both $l' \geq l$ and $l'' \geq l$ are added to $\mathcal{M}$, and l is dropped from G_q. Conceptually, there is a *convergence* in l [12].

   (d) Three AFDs are detected, say $l \rightsquigarrow l'$, $l' \rightsquigarrow l''$, and $l \rightsquigarrow l''$. In this case the three levels in G_q belong to the same hierarchy; $l \geq l' \geq l''$ is added to $\mathcal{M}$ and only the level with highest cardinality, l, is kept in G_q.

A special case is when both $l \rightsquigarrow l'$ and $l' \rightsquigarrow l$ hold, i.e., the relationship between two levels in the group-by set is one-to-one (e.g., if $G_q = \{\mathsf{teams*.name}, \mathsf{teams*.abbreviation}\}$). Here, one of the levels, say l, is kept in G_q while l' is considered as a *descriptive attribute*, which can be picked by the user when formulating a query to improve the quality of the results by giving additional information about l [12].

Note that using AFDs instead of FDs may entail an approximate satisfaction of the disjointness constraint, which in turn can cause summarizability issues. Disjointness is violated when an instance of measure $m \in M_q$ is related to multiple instances of level $l \in G_q$. This may happen when $Types(l) \supseteq \{\text{simple}, \text{array of simple}\}$ and in some documents the array contains more than one value. In this case, a local check on each document can be performed by submitting a query to the document store. The query should select documents where both m and l exist and l is an array of simple, and return the documents for which the size of the array is greater than 1. If some results are returned, disjointness is locally violated; to avoid double counting, a simple solution is to restrict the values of l to simple type. An example of disjointness violation between measure teams*.score and level teams*.name is shown below:

```
"teams" : {"score" : 100, "name" : ["Washington Bullets","WSB"], "city" : "Washington"}
```

Disjointness can also be violated between two levels in the same hierarchy, $l \succeq l'$, when an instance of l is related to more than one instance of l', i.e., when $l \rightsquigarrow l'$ but $l \not\rightsquigarrow l'$. This situation can be detected either locally as seen before or globally using a different query that groups documents by l counting the distinct values of l' to return the documents having a count greater that 1. If some results are returned, disjointness is globally violated. In the following an example of local violation between the levels teams*.name and teams*.city:

```
"teams" : {"score" : 100, "name":"Washington Bullets", "city" : ["Washington","Washington DC"]}
```

and one of global violation:

```
"teams" : {"score" : 100, "name":"Washington Bullets", "city" : "Washington"}
"teams" : {"score" : 107, "name":"Washington Bullets", "city" : "Washington DC"}
```

Even in this case, a basic solution to avoid double counting is to exclude the values for which there is no disjointness (whose relative significance clearly depends on the threshold ϵ used for AFD detection). Note that, in schema-on-write approaches, summarizability can be enforced in presence of disjointness violations by enclosing sophisticated algorithms in the ETL process [40], which unfortunately is unsuitable to a real-time context like ours.

Finally, the completeness constraint is violated when, for some level $l \in G_q$, there is no instance corresponding to one or more instances of a measure $m \in M_q$; from a conceptual point of view, l is classified as *optional* [12]. In a schema-on-write approach this problem is fixed by aggregating all "dangling" measures into an ad-hoc group of l. Similarly, in our schema-on-read approach, these instances are grouped into a null instance of l thus restoring the completeness condition; then for instance, the $ifNull operator of MongoDB could be used to replace null values with an ad-hoc one when executing q.

4.2.2. Recommend Levels

Recommending additional group-by levels is done when originally the user had selected two or three group-by levels, but only one of them was left in G_q after checking the group-by set. For a given candidate level l, it requires to check that the base integrity constraint is met between l and some other level l' in $\mathcal{M}$, and at least a measure $m \in M_q$ is compatible with G_q. The first level found is proposed to the user, who has the choice to use it or to proceed looking for other levels (up to a maximum of three levels in the group-by set).

Example 5. *Let* $G_q = \{\text{teams*.name}, \text{teams*.city}\}$ *and* $M_q = \{\text{teams*.score}\}$. *Since the AFD* teams*.name $\rightsquigarrow$ teams*.city *holds,* teams*.city *is removed from* G_q. *In this case, to recommend levels we may look for AFDs between* teams*.name *and* id*,* date*,* teams*.abbreviation*,* teams*.city*,* teams*.home*. Since* id *has a many-to-many relationship with* teams*.name*, it is recommended to the user as a possible group-by level.*

4.2.3. Check Summarization

The first possible problem with summarization is related to compatibility, which states that measures cannot be aggregated along levels using any aggregation operator [36]. In principle, checking summarization would require knowing the measure category (flow, stock, or value-per-unit), the type of group-by levels (temporal or non-temporal), and the aggregation operator (Sum, Avg, Min, Max, Count, etc.). Knowing the measure category and type of group-by levels could be used to recommend some aggregation operators, but still the user would have to choose among a number of potential options; besides, correctly classifying a measure into flow, stock, or value-per-unit may be hard for non-skilled users. Therefore, we prefer to enable users to directly pick an aggregation operator for each measure in M_q.

The second situation where summarization could be violated is when a measure $m \in M_q$ has finer granularity than another measure $m' \in M_q$, since

m would be double-counted when executing q. In this case, the user is warned that she will get erroneous results, and she may choose to change the aggregation operator for m (e.g., using Min, Max or Avg instead of Sum will give the correct result) or to drop m from M_q.

Example 6. *Let* $M_q = \{\mathsf{teams*.score}, \mathsf{teams*.players*.points}\}$*. These measures have different granularities, so if the user has chosen to aggregate* teams*.score *(which has finer granularity) using Sum she will get double counting. Then, she can either change the aggregation operator for* teams*.score *to Avg or drop* teams*.score *from* M_q*.*

4.2.4. Translate Md-Query

In this section we show how the md-queries expressed using EXODuS (as of Definition 3) can be translated into the native query language of MongoDB. Aggregations in MongoDB can be expressed using the aggregation framework, which offers the ability to group, project, filter, and sort results in different stages. An aggregate MongoDB query consists of a multi-stage pipeline that transforms documents into an aggregated result (`docs.mongodb.com/manual/aggregation/`).

The pseudo-code of the md-query translation is sketched in Algorithm 4. It starts from a well-formed md-query q and a c-schema $\mathcal{S}$, and produces in output a MongoDB query q_m. Algorithm 4 initializes q_m as an aggregate query with the collection name (Line1) and then works as follows. At first, a \$group stage is added at the end of q_m (Line2). For each attribute a in the set A_q of group-by attributes and measures, if the leaf key k of a is nested in an array, function $arraysOf()$ is called to return the arrays in which k is nested (Lines 5—7). To flatten these arrays one \$unwind stage is added for each array at the beginning of q_m (Lines 8—10). Then, if a is a level, we check if it is a temporal one that has been added during MD enrichment; if so, since a does not exist in the collection as an attribute, an \$addFields stage is inserted before the \$group stage to add an attribute for the dimension $dim(a)$ to which a belongs (Lines 12—14). Then, *AddGroup*() adds attribute a to the \$group stage (Line15). Finally, if a is a measure, it is aggregated using $\Sigma(m)$ and added to the \$group stage of q_m (Lines 16—17).

Example 7. *Let us consider for instance the query characterized by* $G_q = \{\mathsf{teams*.players*.player}, \mathsf{year}\}$*,* $M_q = \{\mathsf{teams*.players*.points}\}$*, and* $\Sigma = \text{Sum}$*, which calculates the total score by player and year. The leaf key* player *is*

Algorithm 4 TranslateMdQuery

```
Require: A c-schema $\mathcal{S}$, a (well-formed) md-query $q = (G_q, M_q, \Sigma)$
Ensure: A mongoDB query $q_m$
 1: $q_m \leftarrow Aggregate(rootOf(\mathcal{S}))$
 2: $AddStage(q_m, \text{\$group})$
 3: $A_q \leftarrow G_q \cup M_q$
 4: for all $a \in A_q$ do
 5:     $k \leftarrow keyOf(a)$
 6:     if $k$ is nested in an array then
 7:         for all $array \in arraysOf(k)$ do
 8:             if $array$ has not been added yet then
 9:                 $AddStage(q_m, \text{\$unwind})$
10:                 $Unwind(q_m, array)$
11:     if $a$ is a level then
12:         if $a$ is a temporal level then
13:             $AddStage(q_m, \text{\$addFields})$
14:             $AddAttribute(q_m, a, dim(a))$
15:         $AddGroup(q_m, a)$
16:     else if $a$ is a measure then
17:         $AddGroup(q_m, a, \Sigma(a))$
18: return $q_m$
```

nested in an array, so two $unwind *stages are required to flatten arrays* teams *and* teams.players *respectively. Level* year *is temporal so an* $addFields *stage is required to retrieve the year from dimension* date *using the (native) function* $year. *The corresponding query in MongoDB is then as follows:*

```
db.games.aggregate
( [ {$unwind :
       $teams
    },
    {$unwind :
       $teams.players
    },
    { $addFields:
      { year: { $year: $date },
      }
    },
    {$group:
      { _id : { name: $teams.name, year: $year}
        score : {$sum : $teams.score}
      }
    }
] )
```

4.3. OLAP Enabling

The goal of this phase is to refine the md-schema by incrementally discovering some roll-up relationships, so that the user is enabled to interact with data in an OLAP fashion and iteratively formulate new md-queries. Completely building all the hierarchies would require to mine all AFDs between levels, which would be computationally very expensive. For this reason, we only build local portions of hierarchies for the levels in the group-by set of the previously-formulated md-query q. Specifically, the idea is to mine, for each level $l \in G_q$, the AFDs of either type $l \rightsquigarrow l'$ (to enable a roll-up of l) or $l' \rightsquigarrow l$ (to enable a drill-down of l). Then, if the user applies a roll-up or drill-down, a new md-query q' is formed and the process is iterated to further extend the hierarchies.

At a given time during a user's session, let $\mathcal{M}$ be the current version of the md-schema and q be the last md-query formulated. Then, the search space for mining AFDs includes the levels that are not in G_q and are not involved in roll-up relationships in $\mathcal{M}$. Like in group-by set checking (Section 4.2.1), we take into account level cardinalities and the structural clues provided by the c-schema to avoid useless checks. In addition, we avoid checking transitive AFDs since we explore one hierarchy at a time.

The whole process is described by Algorithm 5; in the following subsections, roll-up and drill-down discovery are detailed.

4.3.1. Discover Roll-ups

Let $l \in G_q$; discovering possible roll-ups for l requires to mine the AFDs of type $l \rightsquigarrow l'$, with $l' \in L \backslash G_q$ and $depth(l) \geqslant depth(l')$. To avoid useless

Algorithm 5 OLAP Enabling

```
Require: An md-schema $\mathcal{M}$, an md-query $q = (G_q, M_q, \Sigma)$ on $\mathcal{M}$, a threshold $\epsilon$ on the strength of AFDs
Ensure: A (refined) md-schema $\mathcal{M}$
 1: $R \leftarrow L \backslash G_q$
 2: for all $l \in G_q$ do
 3:     if $l$ has not been explored yet then
 4:         for all $l' \in R$ do
 5:             if $l \ngeq l'$ then
 6:                 if $depth(l) \geqslant depth(l')$ and $|l'| \leqslant |l|$ then        ▷ Roll-up discovery
 7:                     $CheckAFD(l, l', \epsilon)$
 8:                     if $l \rightsquigarrow l'$ then
 9:                         update $\mathcal{M}$ with $l \succeq l'$
10:             if $l' \ngeq l$ then
11:                 if $depth(l') \geqslant depth(l)$ and $|l| \leqslant |l'|$ then        ▷ Drill-down discovery
12:                     if at least a measure in $M_q$ is compatible with $G_q \backslash \{l\} \cup \{l'\}$ then
13:                         $CheckAFD(l', l, \epsilon)$
14:                         if $l' \rightsquigarrow l$ then
15:                             update $\mathcal{M}$ with $l' \succeq l$
16:         Mark $l$ as explored and notify the user of possible roll-ups and drill-downs
17: return $\mathcal{M}$
```

checks, an AFD whose right-hand side l' has higher cardinality than l is not checked, since it clearly cannot hold (Line6 in Algorithm 5). For each AFD $l \rightsquigarrow l'$ that is found to hold, the md-schema $\mathcal{M}$ is updated by adding $l \succeq l'$ and the user is notified of the ability of performing a roll-up from l.

Example 8. *Let* $G_q = \{\mathsf{id}, \mathsf{teams*.name}\}$ *and* $M_q = \{\mathsf{teams*.score}, \mathsf{teams*.won}, \mathsf{teams*.results.assists}\}$. *The search space for level* id *only includes* date*, since it is the only level for which* $depth(\mathsf{id}) \geqslant depth(\mathsf{date})$*. Since* $|\mathsf{id}| > |\mathsf{date}|$, $\mathsf{id} \rightsquigarrow \mathsf{date}$ *is checked and it is found to hold, so the md-schema is updated with* $\mathsf{id} \succeq \mathsf{date}$*. The search space for* teams*.name *consists of* teams*.abbreviation*,* teams*.city*, and* teams*.home*; by accessing data we find that* $|\mathsf{teams*.name}| = |\mathsf{teams*.abbreviation}| > |\mathsf{teams*.city}| > |\mathsf{teams*.home}|$*. All three AFDs are checked;* $\mathsf{teams*.name} \rightsquigarrow \mathsf{teams*.city}$ *and* $\mathsf{teams*.name} \rightsquigarrow \mathsf{teams*.abbreviation}$ *are found to hold, while* $\mathsf{teams*.name} \rightsquigarrow \mathsf{teams*.home}$ *does not hold. So the games md-schema is updated with* $\mathsf{teams*.name} \succeq \mathsf{teams*.city}$ *and* $\mathsf{teams*.name} \succeq \mathsf{teams*.abbreviation}$.

4.3.2. Discover Drill-Downs

Here the set of candidate levels for checking AFDs of type $l' \rightsquigarrow l$ includes the levels in $L \backslash G_q$ whose depth is greater or equal than l, i.e., such that $depth(l') \geqslant depth(l)$. As in roll-up discovery, an AFD whose right-hand side l has higher cardinality than l' is not checked, since it cannot hold (Line11 in Algorithm 5). If $l' \rightsquigarrow l$ is found to hold, the md-schema $\mathcal{M}$ is updated with $l' \succeq l$ and the user is notified of the ability of drilling down from l.

Note that drilling down from l to l' could produce a new md-query q' whose group-by set, $G_{q'} = G_q \backslash \{l\} \cup \{l'\}$, is not compatible with the granularity of some of the measures. Since checking compatibility is computationally cheaper than checking AFDs (as explained in Section 4.1.2, it can be done based on the partial order of group-by sets, without accessing data), the AFD for a candidate level l' is checked only if $G_{q'}$ is found to be compatible with the granularity of at least one of the measures in $M_{q'}$ (Line12). Of course, if $l' \rightsquigarrow l$ is found to hold and the user decides to drill-down to l', the incompatible measures in $M_{q'}$ must be dropped.

We finally remark that, when $|l| = |l'|$ or $|l| \equiv |l'|$ (due to approximation), both $l \rightsquigarrow l'$ and $l' \rightsquigarrow l$ are checked (during roll-up and drill-down discovery, respectively), since the relationship between l and l' may be one-to-one. In case both AFDs hold, l' is a descriptive attribute for l.

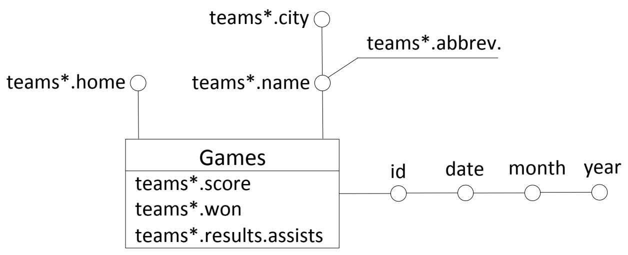

Figure 7: The games md-schema

Example 9. *Consider again Example 8. Since* id *has the highest cardinality, there are no AFDs to check for it. On the other hand, the search space for* teams*.name *consists of* teams*.abbreviation*,* teams*.home *and* teams*.players*.player *(we recall that* teams*.city *has already been added to the hierarchy), where* $|$teams*.home$| < |$teams*.name$| = |$teams*.abbreviation$| < |$teams*.players*.player$|$. *AFD* teams*.home $\rightsquigarrow$ teams*.name *is not checked because of its cardinality; the same for* teams*.players*.player $\rightsquigarrow$ teams*.name*, since the measures in $M_{q'}$ are not compatible with $G_{q'} = \{$*id, teams*.players*.player*$\}$. Eventually,* teams*.abbreviation $\rightsquigarrow$ teams*.name *is checked, giving rise to a descriptive attribute.*

The md-schema resulting after OLAP enabling in Examples 8 and 9 is shown in DFM notation in Figure 7. Note that, at the end of the user's session, md-schemata (which summarize all the knowledge acquired in terms of AFDs) can be stored for reuse and sharing.

5. Experimental Evaluation

In this section we evaluate the performance of EXODuS using three real-world datasets (accessible at [41]):

- **Games**. The working example dataset has been collected by Sports Reference LLC [33]. It contains around 32K nested documents representing NBA games in the period 1985—2013. Each document represents a game between two teams with at least 11 players each. It contains 47 attributes; 40 of them are numeric and represent team and player results.

- **DBLP**. This dataset contains 2M documents scraped from DBLP (`dblp.uni-trier.de/xml/`) in XML format and converted into JSON. Documents are flat and represent eight kinds of publications including conference proceedings, journal articles, books, thesis, etc. The third portion of the dataset represent author pages, containing half the number of attributes compared to other kinds. So, documents have shared attributes such as title, author, type, year and unshared ones such as journal and booktitle.

- **Twitter**. This dataset contains 2M tweets scraped from the Twitter API [38]. Each document represents a tweet message and its metadata, which contains some nested objects: a user object that represent the author of the tweet, a place object that gives its location and a retweet object if it is a reply. The dataset is heterogeneous and mixes between tweets and documents of an API call for tweet deletes.

Table 1 shows the characteristics of the used datasets. Each dataset has been loaded as a collection in a local MongoDB instance running on Intel Core i7 CPU at 3.6 GHz and 8 GB of RAM machine with Ubuntu 14.04 OS.

We have focused testing on the parts of our approach that require accessing data, since the purely algorithmic parts have no relevant computational complexity. Query execution is delegated to MongoDB. Specifically, in the following subsections we evaluate the performance of the three phases of EXODuS.

5.1. MD Enrichment

To evaluate this phase, we measure the time $t_{cSchema}$ required to build the c-schema since it is the only part that requires to access data. In Table 1 we show, for our three datasets, the time required to extract the c-schema and the number of performed iterations. Remarkably, the maximum time

Table 1: Performance of MD enrichment

Dataset	Size	#Documents	#Attr	#Iterations	$t_{cSchema}$
Games	167.5MB	32K	47	3	17.4 sec.
DBLP	683MB	2M	27	1	24.6 sec.
Twitter	4GB	2M	100	5	142.9 sec.

for building a c-schema is less than 143 seconds, which is compatible with real-time usage. We also emphasize that this step is executed only once for each analysis session, at its beginning.

Since also cardinality checks are done once in a session, we discuss them in this section. Table 2 shows for each level its cardinality and the time to retrieve it, t_{card}, which is normally quite small since MongoDB can use indexes instead of collection scans. Note that for levels having high cardinality,

Table 2: Performance of cardinality checks

Dataset	l	$\|l\|$	t_{card}
Games	date	4424	0.0 sec.
	id	31686	0.1 sec.
	teams*.abbreviation	36	0.0 sec.
	teams*.city	31	0.0 sec.
	teams*.home	2	0.0 sec.
	teams*.name	36	0.0 sec.
	teams*.players*.player	2191	0.4 sec.
	all levels	—	**0.6** sec.
DBLP	author	1691491	6.0 sec.
	booktitle	10606	0.0 sec.
	journal	1669	0.0 sec.
	type	8	0.0 sec.
	year	83	0.0 sec.
	all levels	—	**6.1** sec.
Twitter	created_at	377942	0.8 sec.
	place.full_name	7248	0.0 sec.
	place.name	6657	0.0 sec.
	place.country	119	0.0 sec.
	place.country_code	117	0.0 sec.
	source	13519	0.0 sec.
	user.created_at	1550216	5.9 sec.
	user.lang	17	0.0 sec.
	user.location	583070	5.3 sec.
	user.name	1409401	6.3 sec.
	user.screen_name	1569422	5.8 sec.
	all levels	—	**24.4** sec.

the time to retrieve a cardinality is high due to the upper limit of document size in MongoDB (16MB); so we had to use a different query that could not benefit from indexes and suffered from memory limits in MongoDB.

5.2. Querying

Since checking group-by sets requires AFD detection, to evaluate querying we start by measuring the time to check each single AFD possibly present in each md-schema (obviously excluding those whose check can be avoided). Tables 3, 4, and 5 show, for each AFD checked between a pair of levels, its strength and the checking time t_{AFD}. Clearly, t_{AFD} depends on the levels and on their depths. This is apparent for the AFDs involving the author attribute in Table 4, which have a higher execution time. The reason for this is the amount of memory allowed by MongoDB for group-by queries, which is limited to 100 MB; when a query does not fit in memory, temporary files must be written on disk, which dramatically slows execution down. By comparing the figures in Table 2 on the one hand and Tables 3, 4, and 5 on the other we can also confirm that using cardinalities to reduce the search space can significantly improve the overall performance, since retrieving a cardinality is faster than checking an AFD.

Then we consider four md-queries on each dataset (the group-by sets of all twelve queries are shown in Table 6); we assess the performance of query

Table 3: Performance of AFD checks on the Games dataset

$l \rightsquigarrow l'$	$strength(l, l')$	t_{AFD}
id $\rightsquigarrow$ date	100%	0.1 sec.
teams*.name $\rightsquigarrow$ teams*.abbreviation	100%	0.1 sec.
teams*.name $\rightsquigarrow$ teams*.home	50%	0.1 sec.
teams*.name $\rightsquigarrow$ teams*.city	100%	0.1 sec.
teams*.city $\rightsquigarrow$ teams*.home	50%	0.1 sec.
teams*.abbreviation $\rightsquigarrow$ teams*.name	100%	0.1 sec.
teams*.abbreviation $\rightsquigarrow$ teams*.home	50%	0.1 sec.
teams*.abbreviation $\rightsquigarrow$ teams*.city	100%	0.1 sec.
teams*.players*.player $\rightsquigarrow$ teams*.name	34.7%	0.8 sec.
teams*.players*.player $\rightsquigarrow$ teams*.abbrev.	34.7%	0.8 sec.
teams*.players*.player $\rightsquigarrow$ teams*.home	50.5%	0.8 sec.
teams*.players*.player $\rightsquigarrow$ teams*.city	35.1%	0.8 sec.

Table 4: **Performance of AFD checks on the DBLP dataset**

$l \rightsquigarrow l'$	$strength(l, l')$	t_{AFD}
year $\rightsquigarrow$ type	23.5%	1.7 sec.
journal $\rightsquigarrow$ type	99.9%	1.1 sec.
journal $\rightsquigarrow$ year	6.9%	1.2 sec.
booktitle $\rightsquigarrow$ type	67.0%	1.4 sec.
booktitle $\rightsquigarrow$ year	29.3%	1.3 sec.
booktitle $\rightsquigarrow$ journal	no relationship	0.7 sec.
author $\rightsquigarrow$ type	67.0%	15.0 sec.
author $\rightsquigarrow$ year	42.8%	11.9 sec.
author $\rightsquigarrow$ journal	52.9%	6.3 sec.
author $\rightsquigarrow$ booktitle	42.4%	7.3 sec.

validation in the worst case, i.e., assuming that each query is formulated on the draft md-schema first, so that no AFDs previously acquired can be reused. Specifically, in Table 6 we show the time $t_{querying}$ for checking the group-by set of each query (i.e., the sum of the times for the AFD checks required) since it is the only part that requires to access data[3]. The table also shows the number of AFD checks avoided (#Avoided) using cardinalities and structural information from the c-schema. For instance, for query q_1, the AFDs to be considered are the ones between the two attributes in the group-by set, teams*.name and date; of these, date $\rightsquigarrow$ teams*.name is not checked because $depth(\text{date}) < depth(\text{teams*.name})$, while teams*.name $\rightsquigarrow$ date is not checked because $|\text{date}| > |\text{teams*.name}|$. So, overall, 2/2 checks are avoided.

We can conclude that (i) our approach effectively reduces the number of checks by using cardinalities and the c-schema, and (ii) the md-query validation time depends on the number of levels in the group-by set and on their depths. The md-query validation time is very small and fully compatible with real-time usage.

5.3. OLAP Enabling

For OLAP enabling, we consider that each md-query is executed in a session alone. Table 7 shows the overall performance when executing one

[3]The time for actually executing each md-query is not considered here, since the optimization of each md-query is out of the paper scope.

Table 5: Performance of AFD checks on the Twitter dataset

$l \rightsquigarrow l'$	$strength(l, l')$	t_{AFD}
source $\rightsquigarrow$ place.full_name	1,9%	3,5 sec.
source $\rightsquigarrow$ place.name	2%	3,5 sec.
source $\rightsquigarrow$ place.country	29,7%	3,5 sec.
source $\rightsquigarrow$ place.country_code	29,8%	3,5 sec.
source $\rightsquigarrow$ user.lang	82,9%	5,1 sec.
user.location $\rightsquigarrow$ source	74,2%	8,7 sec.
user.name $\rightsquigarrow$ source	90,4%	12,9 sec.
user.created_at $\rightsquigarrow$ source	95,4%	13,3 sec.
user.screen_name $\rightsquigarrow$ source	96,2%	12,1 sec.
created_at $\rightsquigarrow$ source	26,3%	8,7 sec.
created_at $\rightsquigarrow$ place.full_name	95,5%	4,8 sec.
created_at $\rightsquigarrow$ place.name	95,5%	4,8 sec.
created_at $\rightsquigarrow$ place.country	96,5%	4,8 sec.
created_at $\rightsquigarrow$ place.country_code	96,5%	4,9 sec.
created_at $\rightsquigarrow$ user.lang	43,3%	6,9 sec.
user.location $\rightsquigarrow$ created_at	34,2%	9,7 sec.
user.name $\rightsquigarrow$ created_at	73,9%	12,2 sec.
user.created_at $\rightsquigarrow$ created_at	81,2%	13,8 sec.
user.screen_name $\rightsquigarrow$ created_at	82,2%	12,4 sec.
user.name $\rightsquigarrow$ user.location	91,1%	12,9 sec.
user.name $\rightsquigarrow$ place.full_name	96,8%	11,3 sec.
user.name $\rightsquigarrow$ place.name	96,8%	11,4 sec.
user.name $\rightsquigarrow$ place.country	99,3%	11,4 sec.
user.name $\rightsquigarrow$ place.country_code	99,3%	11,3 sec.
user.name $\rightsquigarrow$ user.lang	98,4%	12,2 sec.
user.created_at $\rightsquigarrow$ user.name	98,3%	13,6 sec.
user.screen_name $\rightsquigarrow$ user.name	98,7%	13,2 sec.
place.name $\rightsquigarrow$ place.country	98,6%	3,1 sec.
place.name $\rightsquigarrow$ place.country_code	98,6%	3.0 sec.
place.name $\rightsquigarrow$ user.lang	86,5%	4,7 sec.
place.full_name $\rightsquigarrow$ place.name	100%	3.0 sec.
user.location $\rightsquigarrow$ place.name	76%	6,1 sec.
user.created_at $\rightsquigarrow$ place.name	98,1%	11,8 sec.
user.screen_name $\rightsquigarrow$ place.name	98,1%	10,8 sec.

Table 6: Performance of querying

Dataset	q	G_q	#Avoided	$t_{querying}$
Games	q_1	date, teams*.name	2/2	0.0 sec.
	q_2	id,teams*.name, teams*.home	5/6	0.1 sec.
	q_3	teams*.players*.player, teams*.name, date	5/6	0.8 sec.
	q_4	teams*.players*.player, teams*.name, teams*.home	3/6	1.7 sec.
DBLP	q_5	booktitle, year	1/2	1.3 sec.
	q_6	year, type	1/2	1.7 sec.
	q_7	author, year	1/2	11.9 sec.
	q_8	type, year, author	3/6	28.6 sec.
Twitter	q_9	created_at, source	1/2	8.7 sec.
	q_{10}	source, place.name	1/2	3.5 sec.
	q_{11}	user.name, created_at	1/2	12.2 sec.
	q_{12}	place.name, user.name, created_at	3/6	28.4 sec.

Table 7: Overall performance of OLAP enabling

Dataset	q	#Roll-up, #Drill-down	#Avoided	t_{OLAP}
Games	q_1	1,1	7/10 , 7/10	1.3 sec.
	q_2	2,0	9/12, 7/12	2.2 sec.
	q_3	1,1	6/12, 10/12	2.9 sec.
	q_4	1,0	8/12, 9/12	2.1 sec.
DBLP	q_5	0,0	3/6 , 3/6	24.2 sec.
	q_6	0,1	6/6 , 0/6	31.9 sec.
	q_7	0,0	2/6 , 4/6	32.8 sec.
	q_8	0,1	4/6 , 2/6	18.6 sec.
Twitter	q_9	0,0	8/18 , 10/18	140.4 sec.
	q_{10}	2,1	11/18 , 7/18	130.0 sec.
	q_{11}	2,1	5/18 , 13/18	181.0 sec.
	q_{12}	4,1	10/24 , 14/24	210.8 sec.

OLAP enabling phase (i.e., one roll-up discovery plus one drill-down discovery) starting from each of our twelve md-queries: the number of roll-up and drill-down relationships discovered, the number of checks avoided, and the total time spent, t_{OLAP}. The latter is calculated as the sum of the times for checking each AFD. The results show that the time required by OLAP enabling is reasonable for all md-queries , which confirms that our approach fits real-time contexts.

Example 10. *For query q_1, the AFDs to be considered are those involving the two attributes in the group-by set,* teams*.name *and* date*; namely, for roll-up:*

$\mathsf{date} \rightsquigarrow \mathsf{id}$ *(avoided using cardinalities)*

$\mathsf{date} \rightsquigarrow \mathsf{teams^*.abbreviation}$ *(avoided using the c-schema)*

$\mathsf{date} \rightsquigarrow \mathsf{teams^*.city}$ *(avoided using the c-schema)*

$\mathsf{date} \rightsquigarrow \mathsf{teams^*.home}$ *(avoided using the c-schema)*

$\mathsf{date} \rightsquigarrow \mathsf{teams^*.players^*.player}$ *(avoided using the c-schema)*

$\mathsf{teams^*.name} \rightsquigarrow \mathsf{id}$ *(avoided using cardinalities)*

$\mathsf{teams^*.name} \rightsquigarrow \mathsf{teams^*.abbreviation}$ *(checked)*

$\mathsf{teams^*.name} \rightsquigarrow \mathsf{teams^*.city}$ *(checked)*

$\mathsf{teams^*.name} \rightsquigarrow \mathsf{teams^*.home}$ *(checked)*

$\mathsf{teams^*.name} \rightsquigarrow \mathsf{teams^*.players^*.player}$ *(avoided using the c-schema)*

and for drill-down:

$\mathsf{id} \rightsquigarrow \mathsf{date}$ *(checked)*

$\mathsf{teams^*.abbreviation} \rightsquigarrow \mathsf{date}$ *(avoided using cardinalities)*

$\mathsf{teams^*.city} \rightsquigarrow \mathsf{date}$ *(avoided using cardinalities)*

$\mathsf{teams^*.home} \rightsquigarrow \mathsf{date}$ *(avoided using cardinalities)*

$\mathsf{teams^*.players^*.player} \rightsquigarrow \mathsf{date}$ *(avoided using cardinalities)*

$\mathsf{id} \rightsquigarrow \mathsf{teams^*.name}$ *(avoided using the c-schema)*

$\mathsf{teams^*.abbreviation} \rightsquigarrow \mathsf{teams^*.name}$ *(checked)*

$\mathsf{teams^*.city} \rightsquigarrow \mathsf{teams^*.name}$ *(avoided using cardinalities)*

$\mathsf{teams^*.home} \rightsquigarrow \mathsf{teams^*.name}$ *(avoided using cardinalities)*

$\mathsf{teams^*.players^*.player} \rightsquigarrow \mathsf{teams^*.name}$ *(checked)*

6. Conclusion

In this paper we have proposed EXODuS, an interactive schema-on-read approach for enabling OLAP querying on document stores. To this end, AFDs are incrementally mined to discover MD structures based on the queries of interest for the user. User interaction is mostly limited to the selection of possible dimensions and measures and to a proper choice of the aggregation operators. After validating user queries from the MD point of view and refining the underlying MD schema accordingly, we build local portions of MD hierarchies aimed at enabling OLAP-style user interaction in the form of roll-ups and drill-downs.

Overall, the experiments we conducted show that the performances of our approach are in line with the requirements of a real-time user interaction. However, some relevant issues still need to be explored and are part of our future work. To improve performance, we plan to further optimize the algorithms proposed. Besides, to increase effectiveness, the versioning and evolution of data and schemata in a collection must be considered; we plan to address this issue by searching for temporal AFDs, i.e., for AFDs that are valid not globally but at each instant of time.

References

References

[1] R. Cattell, Scalable SQL and NoSQL data stores, ACM SIGMOD Record 39 (4) (2011) 12–27.

[2] R. Hecht, S. Jablonski, NoSQL evaluation: A use case oriented survey, in: Proc. CSC, 2011, pp. 336–341.

[3] A. Abelló, J. Darmont, L. Etcheverry, M. Golfarelli, J.-N. Mazón, F. Naumann, T. B. Pedersen, S. Rizzi, J. Trujillo, P. Vassiliadis, G. Vossen, Fusion cubes: towards self-service business intelligence, International Journal of Data Warehousing and Mining 9 (2) (2013) 66–88.

[4] A. Cuzzocrea, L. Bellatreche, I.-Y. Song, Data warehousing and OLAP over big data: current challenges and future research directions, in: Proc. DOLAP, 2013, pp. 67–70.

[5] C. Chasseur, Y. Li, J. Patel, Enabling JSON document stores in relational systems, in: Proc. WebDB, 2013, pp. 1–6.

[6] Z. H. Liu, B. Hammerschmidt, D. McMahon, Y. Lu, H. J. Chang, Closing the functional and performance gap between SQL and NoSQL, in: Proc. SIGMOD, 2016, pp. 227–238.

[7] W. Spoth, B. Sadat Arab, E. S. Chan, D. Gawlick, A. Ghoneimy, B. Glavic, B. Hammerschmidt, O. Kennedy, S. Lee, Z. H. Liu, X. Niu, Y. Yang, Adaptive schema databases, in: Proc. CIDR, 2017, pp. 1–9.

[8] Z. H. Liu, D. Gawlick, Management of flexible schema data in RDBMSs - opportunities and limitations for NoSQL, in: Proc. CIDR, 2015.

[9] S. Rizzi, E. Gallinucci, M. Golfarelli, O. Romero, A. Abelló, Towards exploratory OLAP on linked data, in: Proc. SEBD, 2016, pp. 86–93.

[10] A. Löser, F. Hueske, V. Markl, Situational business intelligence, in: Proc. BIRTE, 2009, pp. 1–11.

[11] P. Bourhis, J. L. Reutter, F. Suárez, D. Vrgoč, JSON: data model, query languages and schema specification, in: Proc. PODS, 2017.

[12] M. Golfarelli, D. Maio, S. Rizzi, The dimensional fact model: A conceptual model for data warehouses, International Journal of Cooperative Information Systems 7 (2-3) (1998) 215–247.

[13] C. Phipps, K. C. Davis, Automating data warehouse conceptual schema design and evaluation, in: Proc. DMDW, 2002, pp. 23–32.

[14] C. S. Jensen, T. Holmgren, T. B. Pedersen, Discovering multidimensional structure in relational data, in: Proc. DaWaK, 2004, pp. 138–148.

[15] N. Prat, J. Akoka, I. Comyn-Wattiau, A UML-based data warehouse design method, Decision Support Systems 42 (3) (2006) 1449–1473.

[16] O. Romero, A. Abelló, A framework for multidimensional design of data warehouses from ontologies, Data and Knowledge Engineering 69 (11) (2010) 1138–1157.

[17] O. Romero, A. Abelló, Automatic validation of requirements to support multidimensional design, Data and Knowledge Engineering 69 (9) (2010) 917–942.

[18] M. Golfarelli, S. Rizzi, B. Vrdoljak, Data warehouse design from XML sources, in: Proc. DOLAP, 2001, pp. 40–47.

[19] M. R. Jensen, T. H. Møller, T. B. Pedersen, Specifying OLAP cubes on XML data, Journal of Intelligent Information Systems 17 (2-3) (2001) 255–280.

[20] B. Vrdoljak, M. Banek, S. Rizzi, Designing web warehouses from XML schemas, in: Proc. DaWaK, 2003, pp. 89–98.

[21] M. Golfarelli, S. Graziani, S. Rizzi, Starry vault: Automating multidimensional modeling from data vaults, in: Proc. ADBIS, 2016, pp. 137–151.

[22] K. Dehdouh, Building OLAP cubes from columnar NoSQL data warehouses, in: Proc. MEDI, 2016, pp. 166–179.

[23] P. Zhao, X. Li, D. Xin, J. Han, Graph cube: on warehousing and OLAP multidimensional networks, in: Proc. SIGMOD, 2011, pp. 853–864.

[24] F. D. Tria, E. Lefons, F. Tangorra, Evaluation of data warehouse design methodologies in the context of big data, in: Proc. DaWaK, 2017, pp. 3–18.

[25] M. L. Chouder, S. Rizzi, R. Chalal, Enabling self-service BI on document stores, in: Proc. EDBT/ICDT Workshops, 2017.

[26] A. Abelló, O. Romero, T. B. Pedersen, R. Berlanga, V. Nebot, M. J. Aramburu, A. Simitsis, Using semantic web technologies for exploratory OLAP: A survey, IEEE Transactions on Knowledge and Data Engineering 27 (2) (2015) 571–588.

[27] J. Varga, A. Vaisman, O. Romero, L. Etcheverry, T. B. Pedersen, C. Thomsen, Dimensional enrichment of statistical linked open data, Journal of Web Semantics 40 (2016) 22–51.

[28] M. Armbrust, A. Ghodsi, M. Zaharia, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, Spark SQL: Relational data processing in Spark, in: Proc. SIGMOD, 2015, pp. 1383–1394.

[29] D. S. Ruiz, S. F. Morales, J. G. Molina, Inferring versioned schemas from NoSQL databases and its applications, in: Proc. ER, 2015, pp. 467–480.

[30] M. Klettke, U. Störl, S. Scherzinger, Schema extraction and structural outlier detection for JSON-based NoSQL data stores, in: Proc. BTW, 2015, pp. 425–444.

[31] L. Wang, O. Hassanzadeh, S. Zhang, J. Shi, L. Jiao, J. Zou, C. Wang, Schema management for document stores, VLDB Journal 8 (9) (2015) 922–933.

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

[33] K. Valeri, Crunching 30 years of NBA data with MongoDB aggregation, https://thecodebarbarian.wordpress.com (2014).

[34] J. N. Mazón, J. Lechtenbörger, J. Trujillo, A survey on summarizability issues in multidimensional modeling, Data and Knowledge Engineering 68 (12) (2009) 1452–1469.

[35] M. Golfarelli, S. Rizzi, P. Biondi, myOLAP: An approach to express and evaluate OLAP preferences, IEEE Trans. Knowl. Data Eng. 23 (7) (2011) 1050–1064.

[36] H. J. Lenz, A. Shoshani, Summarizability in OLAP and statistical databases, in: Proc. SSDBM, 1997, pp. 132–143.

[37] Y. Huhtala, J. Kärkkäinen, P. Porkka, H. Toivonen, TANE: An efficient algorithm for discovering functional and approximate dependencies, Computer Journal 42 (2) (1999) 100–111.

[38] M. Discala, D. J. Abadi, Automatic generation of normalized relational schemas from nested key-value data, in: Proc. SIGMOD, 2016, pp. 295–310.

[39] I. F. Ilyas, V. Markl, P. Haas, P. Brown, A. Aboulnaga, CORDS: Automatic discovery of correlations and soft functional dependencies, in: Proc. SIGMOD, 2004, pp. 647–658.

[40] T. B. Pedersen, C. S. Jensen, C. E. Dyreson, Extending practical pre-aggregation in on-line analytical processing, in: Proc. VLDB, 1999, pp. 663–674.

[41] M. L. Chouder, S. Rizzi, R. Chalal, JSON datasets for exploratory OLAP, http://dx.doi.org/10.17632/ct8f9skv97.1 (2017).