Enabling Self-Service BI on Document Stores

Mohamed L. Chouder
LMCS, ESI
Algiers,Algeria
m_chouder@esi.dz

Stefano Rizzi
DISI, University of Bologna
Bologna, Italy
stefano.rizzi@unibo.it

Rachid Chalal
LMCS, ESI
Algiers,Algeria
r_chalal@esi.dz

ABSTRACT

The growing use of document stores has resulted in vast amounts of semi-structured data holding precious information, which could be profitably integrated into existing BI systems. Unfortunately, due to their schemaless nature, document stores are hardly accessible via direct OLAP querying. In this paper we propose an interactive, schema-on-read approach for finding multidimensional structures in document stores aimed at enabling OLAP querying in the context of self-service BI. Our approach works in three steps: multidimensional enrichment, querying, and OLAP enabling; the validation of user queries and the detection of multidimensional structures is based on the mining of approximate functional dependencies from data. The efficiency of our approach is discussed with reference to real datasets.

CCS Concepts

•Information systems → Data warehouses; •Applied computing → Business intelligence;

Keywords

NoSQL, Document-Oriented Databases, JSON, Multidimensional modeling

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 many implementations [4]. They can be classified based on their data model, the most popular categories being key-value, wide-column, graph-based and document-oriented; document-oriented databases (briefly, *document stores*) are probably the most widely adopted so far. The common features of document stores are *horizontal scalability* on commodity hardware and the lack of an explicit schema (according to the *data first, schema later or never* paradigm). This *schemaless* nature provides a flexible data model that has attracted developers seeking to avoid the restrictions posed by the relational model.

Document stores usually collect nested, denormalized, and hierarchical documents in the form of collections; the documents within the same collection may present a structural variety due to schema flexibility and data evolution. Documents are self-describing and mainly encoded using the semi-structured data format JSON (JavaScript Object Notation), which is also popular as an exchange format and widely adopted in modern web APIs.

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 [1]. Indeed, multidimensional modeling and analysis has been recognized to be an effective way for conducting analytics over big NoSQL data [6]. Unfortunately, due to their schemaless nature, document stores are hardly accessible via direct OLAP querying. Recent efforts in this area propose SQL-like query interfaces (e.g., Apache Drill, Spark SQL, Presto), which provide access to schemaless and nested data while offering the possibility of using traditional BI tools. However, none of these solutions supports multidimensional querying or OLAP over document stores.

In this paper we propose an interactive, schema-on-read approach for finding multidimensional structures in document stores aimed at enabling OLAP querying in the context of self-service BI. Differently from a *schema-on-write* approach, which would force a fixed structure in data and load them into a data warehouse, a *schema-on-read* approach leaves data unchanged in their structure until they are accessed by the user [16]. We claim that, when querying document stores for self-service BI, a schema-on-read approach should be preferred to a schema-on-write approach. The main reason for this is that document stores handle high volumes of data with varied structure, which question the ability of the 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 hand.

To discover the information necessary for discovering multidimensional concepts despite the lack of structure, we resort to data distributions, i.e., we mine approximate functional dependencies (AFDs) and use them to automate the design process. More specifically, our approach takes as input a collection of nested documents and operates in three phases.

1. **Multidimensional Enrichment**. In this phase, a schema is extracted out of the collection and enriched with some basic multidimensional knowledge. This is done by tentatively classifying attributes into dimensions and measures, and by relating each measure to the subset of dimensions that determine its granularity, so as to create a draft multidimensional schema.

2. **Querying**. Now the user can formulate a multidimensional query by picking dimensions and measures of interest. This query is checked by accessing data to ensure its multidimensional validity and correct summarization. If the query is found to be well-formed, it is executed. Otherwise, the multidimensional schema is refined and some possible querying alternatives are proposed to the user.

3. **OLAP Enabling**. This iterative phase enables the user to further explore data by running an OLAP session. To this end, local portions of multidimensional hierarchies (consisting of roll-up and drill-down relationships for each level involved in the user query) are built by mining AFDs from data. The user can now apply an OLAP operator (e.g., roll-up or drill-down) to iteratively create a new query on the collection.

The advantages of the proposed approach are twofold. First, it enables decision makers to formulate multidimensional queries and create reports on-the-fly on document stores in a self-service fashion, i.e., without any support from ICT people. This reduces the time and effort spent in traditional BI settings. Second, our approach 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`). The approach has been implemented on top of MongoDB, one of the most popular document stores.

The rest of this paper is organized as follows. Section 2 discusses the related work and Section 3 presents document stores along with the working example. Our approach is detailed in Section 4 and experimentally evaluated in Section 5. Section 6 concludes the paper and gives some future directions for research.

2. RELATED WORK

Supply-driven multidimensional design. The automation of multidimensional modeling has been widely explored in the area of data warehouse design. Supply-driven approaches automate the discovery of multidimensional concepts by a thorough analysis of the source data. The first approaches proposed algorithms to build multidimensional schemata starting from Entity/Relationship diagrams or relational schemata [8, 19, 12], while further approaches considered more expressive conceptual sources such as UML class diagrams [20] and ontologies [23]. Furthermore, [24] discussed the use of FDs for automating multidimensional modeling from ontologies. In particular, similarly to the querying phase of our approach, [22] proposes an algorithm to check the multidimensional validity of an SQL cube query and to derive the underlying multidimensional schema. This algorithm identifies multidimensional concepts from the structure of the query itself and by following foreign keys in the relational schema.

In these approaches, multidimensional 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 multidimensional concepts by mining FDs from data, as required by the schemaless context.

Multidimensional design from non-relational data. Our approach closely relates to previous approaches for multidimensional modeling from semi-structured data [9, 13, 28] 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 [28] builds multidimensional 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* [7] is a recent approach that mines FDs for multidimensional 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 multidimensional 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 intelligently reduce the search space.

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 [2]. Some works in this direction address the problem of building multidimensional hierarchies from linked data [21, 27], which is closely related to the third phase of our approach. Specifically, *iMOLD* [21] is an instance-based approach that operates at query time for exploratory OLAP on linked data; it aims at finding multidimensional 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, it 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, 17] (e.g., Oracle, 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 [17]. 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-like interfaces that offer extensions to query schemaless data persisted in document stores or as JSON files (e.g., in Hadoop). Spark SQL [3] 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 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 connect to MongoDB, which gave us various architectural options for implementation. However, none of them supports multidimensional or OLAP querying over document stores.

3. DOCUMENT STORES

A document store holds a set of databases, each organizing the storage of documents in the form of *collections* in a way similar to rows and tables in relational databases. A *document*, also called *object*, consists of a set of key-value pairs (or fields) mainly encoded in JSON format (`www.json.org`). JSON 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 [17]. Keys are always strings, while values have the following types: *primitive* (number, string, Boolean), *object* (or *sub-document*), *array* of atomic values or objects.

From a conceptual point of view, a typical collection consists of a set of business entities 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 the other hand, references are similar to foreign keys in relational databases and 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. Since a document store has a flexible schema there are a variety of ways for data representation; nevertheless, the way data is modeled may affect performance and data retrieval patterns.

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

(a)

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

(b)

Figure 1: Sample JSON document of a game

Example 1. *In the scope of this paper, we focus on collections of JSON documents that logically represent the same business entities, while expecting that their structure may vary. In particular, we will use as a working example a collection that models NBA games (adapted from [26]), where a document represents one game between two teams along with players and team results. In the sample document shown in Figure 1a the main object is* games*, which holds the array of objects* teams *(to-many), which in turn holds the object* results *(to-one) and the array of objects* players *(to-many). Figure 1b shows an alternative JSON representation of the same domain; the relationship between game and teams is modeled using two separate sub-documents (home and guest team) instead of an array. Note that a simple query that computes the average* score *by team would be much easier to implement using the first representation and it would also perform better, since in the second representation it would require two separate queries.*

4. APPROACH

As already discussed, our approach for enabling self-service BI on document stores includes three phases: multidimensional enrichment, querying, and OLAP enabling; these phases are discussed in the detail in the following subsections.

4.1 Multidimensional Enrichment

The aim of this phase is to define a draft multidimensional schema on which the user will be able to formulate queries. To this end, we extract the schema of the input collection and enrich it with basic multidimensional knowledge. Several works have addressed schema extraction and discovery from document stores [25, 29, 14]. These works focus on the structural variety of documents within the same collection caused by their schemaless nature and by the evolution of data; for instance, [14] proposes to extract a JSON schema (`www.json-schema.org`) similar to a JSON document and infer attribute and type occurrence frequencies to emphasize the structural variety. In [29], all the schema versions in a collection are extracted and stored in a repository to be used within a schema management framework. In order to have a single view of the distinct schemata in a collection, the authors propose a relaxed form of schema called *skeleton* that better captures the core and prominent fields and filters out unfrequent ones. This would be useful for us since we rely on a single view of the collection; however, the computational cost for building skeletons is high, making them unsuitable in real-time contexts.

Since schema discovery is not the focus of this paper, we adopt a simple algorithm that builds a tree-like schema including all the fields that appear in the documents [3]. This algorithm works in a single-pass over data to infer a tree structure of fields and their corresponding types (fields with varying types are generalized to String).

The tree-like collection schema that will be the input for the subsequent steps of our approach is 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 distinguished root node and is labelled with the collection name;*
- *E is a set of arcs that includes (i) an arc $r \to k$ for each key/value pair in the root r, and (ii) an arc $k_1 \to k_2$ iff k_2 appears as a key in an array of nested objects having key k_1.*

Figure 2 shows the c-schema to which the document of Figure 1a conforms. A path from the root key to a leaf key is called an *attribute*. In denoting attributes we use the dot notation omitting the root key (since a single collection is involved) but including the keys corresponding to nested objects; so, for instance, the c-schema in Figure 2 contains attributes id, date, teams.name, teams.results.points, teams.players.player, teams.players.points, etc. The depth of an attribute a is the number of arcs included in its path, and is denoted by $depth(a)$ (e.g., $depth(\text{id}) = 1$); the set of attributes at depth δ is denoted by $Attrs(\delta)$.

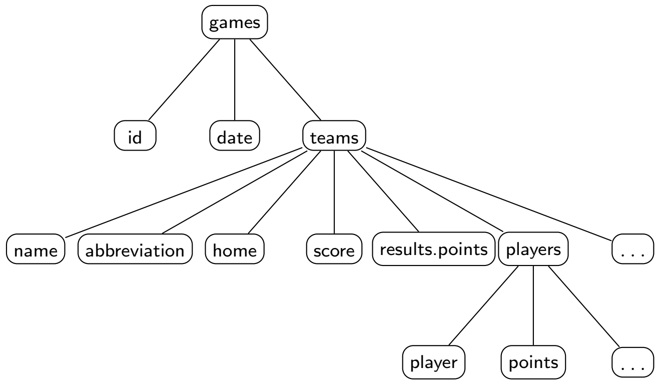

Figure 2: NBA games c-schema

After the c-schema $\mathcal{S}$ of a collection has been defined, a corresponding multidimensional 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 dimension.

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. Each element $G \in 2^L$, where $L = \bigcup_i L_i$ and at most one level in G is taken from each hierarchy h_i, 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 its* dimensions*, i.e., 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 dimensions (i.e., as levels of different hierarchies), and all attributes of types integer and decimal as measures. Date dimensions are then decomposed into different categorical levels giving rise to standard temporal hierarchies (e.g., date $\succeq$ month $\succeq$ year). Note that no further attempt to discover hierarchies is made at this stage, since discovering the FDs involving *all* attributes would be computationally too 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 $l \succeq l'$.

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 dimension, 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, the second situation requires that the values of the non-numeric attribute are transformed into numerical values to be supported by aggregation functions.

To complete the definition of $\mathcal{M}_{draft}$, the granularity mapping g must be built. We recall from [22] that an md-schema should comply with the *multidimensional space arrangement constraint*, stating that each instance of a measure is related to one instance of each dimension. 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. 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 $depth(m) \geqslant depth(l)$.

EXAMPLE 2. *In this example and in the following ones we will refer to the representation in Figure 1a. Attribute* teams.score *is numerical, so it is assumed to be a measure. It is $depth(\text{teams.score}) = 2$, therefore* teams.score *is tentatively associated in $\mathcal{M}_{draft}$ with group-by set $G = \{\text{teams.name}, \text{teams.abbreviation}, \text{teams.home}, \text{id}, \text{date}\}$. All the measures that have the same depth of* teams.score *(e.g.,* teams.won *and* teams.results.points*) are also associated with G. Similarly, measure* teams.players.points *has depth 3, so it is associated with group-by set* $G' = \{$teams.players.player, teams.name, teams.abbreviation, teams.home, id, date$\}$.

4.2 Querying

This phase is aimed at supporting a non-expert user in formulating a well-formed multidimensional 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 multidimensional 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*

- *$G_q \in \mathcal{G}$ 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 $\mathcal{M}_{draft}$, the user formulates an md-query q by choosing one to three dimensions (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 (and the md-schema q is formulated on) should comply with the following constraints [22]:

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

$\sharp 2$ 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) [15].

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

Example 3. *A possible md-query on the draft md-schema described in Example 2 is the one characterized by $G_q = \{$*teams.players.player, date*$\}$, $M_q = \{$*teams.score*$\}$, and $\Sigma =$* Sum. *Unfortunately, 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). So, this query violates disjointness and is not well-formed.*

The pseudo-code of the querying phase is sketched in Algorithm 1. It starts from the md-query q and takes as input both the c-schema and the md-schema. The output of the algorithm is a valid md-query based on a refined md-schema to be used in the next phase. Algorithm 1 works as follows. Firstly, procedure *CheckMeasures* drops from M_q the measures, if any, that are not compatible with G_q based on their granularity (Line 1). If no measure is left, some alternative (compatible) measures are suggested by *RecommendMeasures* (Lines 2–5). If some measures M_{rec} are found, the user interacts to approve their inclusion in the md-query (Line 5). Then, if the md-query group-by set includes either 2 or 3 levels (Lines 7–11), 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 8). If just one level is left in G_q, procedure *RecommendLevels* is called to look for additional group-by levels (Line 10) and possibly include them in the md-query (Line 11). Finally, procedure *CheckSummarization* checks the aggregation operators (Line 12). We allow the resulting md-query to be executed with a single group-by level, but not without measures (Lines 6 and 13); in the latter case, the user is asked to choose a new group-by set (Line 15).

Algorithm 1 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: $M_q \leftarrow CheckMeasures(M_q, G_q, \mathcal{M})$
 2: if $M_q = \varnothing$ then
 3:     $M_{rec} = RecommendMeasures(G_q, \mathcal{M})$
 4:     if $M_{rec} \neq \varnothing$ then
 5:         update $M_q$ based on the user's choice
 6: if $M_q \neq \varnothing$ then
 7:     if $|G_q| > 1$ then
 8:         $G_q = CheckGroupBy(M_q, G_q, \mathcal{M}, \mathcal{S})$
 9:         if $|G_q| = 1$ then
10:             $L_{rec} = RecommendLevels(G_q, \mathcal{M})$
11:             update $G_q$ and $\mathcal{M}$ based on the user's choice
12:     $\Sigma \leftarrow CheckSummarization(q, \mathcal{M})$
13:     $Execute(q)$
14: else
15:     ask the user to change the md-query group-by set $G_q$
```

4.2.1 Checking Measures

The goal of this procedure is to ensure that all the measures in M_q can be correctly aggregated at group-by set G_q; specifically, our goal is to avoid that for some $m \in M_q$ there is a to-many relationship between m and a level $l \in G_q$, because this would violate disjointness and lead to double counting when executing q. Measure check is based on the g component of $\mathcal{M}$, which expresses the granularity of each measure.

To explain how this is done, we observe that the roll-up partial orders on the hierarchies H of $\mathcal{M}$ induce a partial order $\succeq_H$ on the set $\mathcal{G}$ of the group-by sets of H, defined as the product order of the single roll-up orders.[1] Measure m 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, meaning that m can be safely aggregated at G_q. The measures that are found not to be compatible with G_q are removed from M_q.

Example 4. *Considering again the query in Example 3, it is $g($*teams.score*$) \not\succeq_H G_q$ (because* teams.players.player *is not in $g($*teams.score*$)$). Then, q is not well-formed and* teams.score *is dropped from M_q.*

[1]The product order of n total orders is a partial order on the Cartesian product of the n totally ordered sets, such that $(x_1, \ldots, x_n) \succeq (y_1, \ldots, y_n)$ iff $x_i \succeq y_i$ for $i = 1, \ldots, n$.

This check is performed at the schema level, so some exceptions may arise when the relationship between a measure m and G_q is hidden in data. This may happen when arrays are not used to model to-many relationships, or when m depends either on a level not present in G_q or on a subset of the levels in G_q [18]. In these cases, m may be non-additive or even non-aggregable on G_q, so the user's knowledge of the application domain is required to manually fix summarization (see Section 4.2.5).

4.2.2 Checking Group-by Set

This process is aimed at ensuring completeness and base integrity.

The completeness condition is violated when, for some level $l \in G_q$, there is no instance corresponding to one or more instances of a measure in M_q; from a conceptual point of view, l is classified as *optional* [8]. 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.

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, for each pair of levels $l, l' \in G_q$, there is no FD between l and l', i.e., there is a many-to-many relationship between them. An FD is a to-one relationship, which we will denote with $l \rightarrow l'$ to emphasize that values of l functionally determine the values of l'. Checking base integrity requires to access data in order to retrieve AFDs between the levels, in G_q, that are not related by a to-many multiplicity in the c-schema (see Section 4.3 for more details about AFDs).

We recall that $1 \leqslant |G_q| \leqslant 3$:

1. If $|G_q| = 1$, orthogonality is obvious.
2. If $|G_q| = 2$, the relationship between the two levels in G_q is checked. If it turns out to be many-to-many (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 (e.g., if $G_q = \{\mathsf{id}, \mathsf{date}\}$), then there is a roll-up relationship between the two levels in G_q. Only the level at the "many" side (id) remains in G_q and the md-schema is modified by placing these levels in the same hierarchy ($\mathsf{id} \geq \mathsf{date}$). A special case is when the relationship is one-to-one (e.g., if $G_q = \{\mathsf{teams.name}, \mathsf{teams.abbreviation}\}$); here, one of the levels is kept in G_q while the other is considered as a descriptive attribute.
3. In case $|G_q| = 3$, there are three possibilities:
   (a) If many-to-many relationships are found between each pair of levels, then the 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) Let many-to-many relationships be found between two pairs of levels (l, l'), (l, l''), and a many-to-one relationship be found between the remaining pair (l', l'') (e.g., if $G_q = \{\mathsf{teams.name}, \mathsf{id}, \mathsf{date}\}$, since $\mathsf{id} \rightarrow \mathsf{date}$). In this case, $l'$and l'' are placed in the same hierarchy in the md-schema ($l' \geq l''$), and l'' is removed from G_q.
   (c) Finally, when $l' \rightarrow l$ and $l'' \rightarrow l$,
      i. if the relationship between l' and l'' is many-to-many, they are both retained in G_q; conceptually, l is a *convergence* [8] and the md-schema must be updated by adding $l' \geq l$ and $l'' \geq l$.
      ii. if the relationship between l' and l'' is many-to-one, then the three levels in G_q belong to the same hierarchy ($l' \geq l'' \geq l$) and only the level with highest cardinality, l', is kept in G_q.

4.2.3 Recommending Measures

When there is no compatible measure left in M_q, this procedure looks for alternative measures. Specifically, it returns all the measures in the md-schema that are compatible with G_q, i.e., all $m \in M$ such that $g(m) \geq_H G_q$.

EXAMPLE 5. *With reference to Example 4, procedure RecommendMeasures returns* teams.players.points*, since it is $g(\mathsf{teams.players.points}) \geq_H G_q$.*

4.2.4 Recommending 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 in $\mathcal{M}$, and that the measure check is not violated. 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 6. *Let $G_q = \{\mathsf{id}, \mathsf{date}\}$ and $M_q = \{\mathsf{teams.players.points}\}$. The relationship between the levels in G_q is many-to-one, so* date *is removed. In this case, to recommend levels we may look for a many-to-many relationship between* id *and* teams.name*,* teams.abbreviation*,* teams.home*, and* teams.players.player*. Since* teams.name *does not violate the measure check and has a many-to-many relationship with* id*, it is recommended to the user as a possible group-by level.*

4.2.5 Checking Summarization

Compatibility states that measures cannot be aggregated along hierarchies using any aggregation operators. 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.) [15]. 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.

Summarization is violated 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 7. *Let* $M_q = \{\mathsf{teams.score}, \mathsf{teams.players.pts}\}$*. 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*.*

4.3 OLAP Enabling

The goal of this phase is to refine the md-schema by discovering some hierarchies, so that the user is enabled to interact with data in an OLAP fashion. Completely building all the hierarchies would require to mine all FDs between levels, which would be computationally too 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. Besides, again for complexity reasons, we only mine simple FDs (i.e., those relating single attributes rather than attribute sets), which are mostly common in multidimensional schemata.

Specifically, the idea is to mine, for each level $l \in G_q$ for which no roll-up relationships have been discovered yet, the FDs of either type $l \to l'$ (to enable a roll-up of l) or $l' \to l$ (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. Remarkably, using FDs to discover hierarchies guarantees that q' satisfies the conditions discussed earlier, specifically disjointness and completeness.

FDs are not explicitly modeled in a document store, so we must resort to data. Since document stores host large amounts of data, we can reasonably assume that the mined FDs are representative enough of the application domain. However, schemaless data commonly present some errors and missing values which may hide some FDs. The tool we adopt to cope with this issue are *approximate* FDs (AFDs) [10, 11], which "almost hold" on data, instead of traditional ones. Unfortunately, this entails an approximate satisfaction of the disjointness and completeness conditions, possibly leading to non-disjoint and incomplete hierarchies.

DEFINITION 4 (APPROXIMATE FD). *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'*. We will say that AFD* $l \rightsquigarrow l'$ *holds if* $strength(l, l') \geqslant \epsilon$*, where* ϵ *is a user-defined threshold* [11].

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 $\mathcal{M}$. To reduce the search complexity we take into account the structural clues provided by the c-schema. Specifically, let l and l' be two levels, such that $depth(l) < depth(l')$; as already stated, the attributes in $Attrs(\delta)$ are related by to-many multiplicity to those in $Attrs(\delta + 1)$, so $l \not\rightsquigarrow l'$. For this reason, checking if l and l' should be placed in the same hierarchy only requires to check if $l' \rightsquigarrow l$. In addition, we avoid checking trivial and transitive AFDs since we explore one hierarchy at a time. Finally, the result of all AFD checks performed is stored in a meta-data repository, to be used to avoid checking the same AFDs twice.

In the following subsections, roll-up and drill-down discovery are detailed.

4.3.1 Roll-up Discovery

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 $depth(l') \leqslant depth(l)$. Let R be the set of all right-hand sides for these AFDs. To avoid useless checks, the AFDs whose right-hand side l' has higher cardinality than l are not checked, since they clearly cannot hold. One exception is when $|l| = |l'|$ or $|l| \gtrsim |l'|$ (due to approximation): in this case both $l' \rightsquigarrow l$ and $l \rightsquigarrow l'$ must be checked, since the relationship between l and l' may be one-to-one (so l' is a descriptive attribute for l). These bidirectional checks are made at the end of the process to prioritize the discovery of "true" hierarchies. The AFDs in R are checked until one is found to hold, say $l \rightsquigarrow l'$. Then, the md-schema $\mathcal{M}$ is updated by placing these levels in the same hierarchy ($l \geq l'$) 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* $M_q = \{\mathsf{teams.score}, \mathsf{teams.won}\}$*. The search space for level* id *only includes* date*, since it is the only level for which* $depth(\mathsf{date}) \leqslant depth(\mathsf{id})$*. Since* $|\mathsf{id}| > |\mathsf{date}|$*,* $\mathsf{id} \rightsquigarrow \mathsf{date}$ *is checked. This is found to be true, so the md-schema is updated with* $\mathsf{id} \geq \mathsf{date}$*. The search space for* teams.name *consists of* teams.abbreviation *and* teams.home*; by querying data we find that* $|\mathsf{teams.name}| = |\mathsf{teams.abbreviation}| > |\mathsf{teams.home}|$*. Firstly,* $\mathsf{teams.name} \rightsquigarrow \mathsf{teams.home}$ *is checked, and found not to hold. Then,* $\mathsf{teams.name} \rightsquigarrow \mathsf{teams.abbreviation}$ *and* $\mathsf{teams.abbreviation} \rightsquigarrow \mathsf{teams.name}$ *are checked; both are found to hold, giving rise to a descriptive attribute.*

4.3.2 Drill-Down Discovery

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 left-hand side l' has lower cardinality than l is not checked, since it cannot hold. If $l' \rightsquigarrow l$ is found to hold, $l' \geq l$ is added to $\mathcal{M}$ 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 some of the measures in $M_{q'} = M_q$. Since checking compatibility is computationally cheaper than checking AFDs (as explained in Section 4.2.1, 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 at least one of the measures in $M_{q'}$. 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.

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

The md-schema resulting after OLAP enabling in Examples 8 and 9 is shown in DFM notation [8] in Figure 3. Note that, at the end of the user's session, md-schemata can be stored for reuse and sharing.

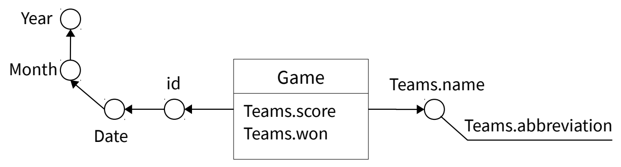

Figure 3: The games md-schema

5. EXPERIMENTAL EVALUATION

We evaluate the performance of our approach using two real-world datasets :

- **Games**. The working example dataset has been collected by Sports Reference LLC [26]. It contains around 32K nested documents representing NBA games in the period of 1985–2013. Each document represents a game between two teams with at least 11 players each. It contains 46 attributes; 40 of them are numeric and represent team and player results.
- **DBLP**. This dataset contains 5.4M 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. So, they have common attributes such as title, author, type, year and uncommon ones such as journal and booktitle.

Implementation. Each dataset has been loaded as a collection in a local MongoDB instance running on Intel Core i5 CPU at 3.3 GHz and 4 GB of RAM machine with Ubuntu 14.04 OS. We have focused on the parts of our approach that require accessing data to retrieve multiplicities of inter-level relationships by means of COUNT DISTINCT queries, since the purely algorithmic parts have no relevant computational complexity. Query execution can be delegated either to the document store (MongoDB in our architecture) or to query engines such as Spark SQL, Drill, and Presto. Based on the tests we performed, these engines are significantly slower than native querying, plausibly because of the different query execution plans. In addition, these engines sometimes return incorrect results when it comes to COUNT DISTINCT queries involving nested attributes. Therefore, we defined two queries based on the native query language of MongoDB:

1. The first one (Q_{AFD}) checks the AFD between a pair of levels and returns its approximation using the formula of Definition 4.
2. The second one (Q_{card}) returns the cardinality of a level.

Methodology. In the previous section, we showed how our approach validates md-queries and iteratively derives an md-schema that satisfies the multidimensional constraints. In this section, we consider five md-queries on the *games* dataset, q_1 to q_5, and four on the DBLP dataset, q_6 to q_9 (the group-by sets of all queries are shown in Table 1). Then, we measure the time required by the querying and OLAP-enabling phases. We also discuss the efficiency of our algorithm based on the number of performed and skipped checks in each phase. The results obtained are proposed below.

5.1 Querying

To evaluate the efficiency of the validation of an md-query, we measure the time of the group-by check since it is the only step that requires to access data.[2] In Table 1 we show, for our nine md-queries, the number of checks avoided using the c-schema (#Avoided), and the total time for validating the md-query ($t_{querying}$). In order to measure $t_{querying}$, the AFD checks required by a single md-query are executed in parallel. We can easily conclude that (i) our approach effectively reduces the number of checks by relying on the dataset structure, and (ii) the md-query validation time depends on the number of levels in the group-by set and on their depths. Remarkably, the md-query validation time is still reasonable (max 50 secs) even with a large dataset, which makes our approach suitable for real-time usage.

Dataset	q	G_q	#Avoided	$t_{querying}$
Games	q_1	id, teams.name	1/2	0.181
	q_2	date, teams.name	1/2	0.177
	q_3	id,teams.name, teams.home	2/6	0.211
	q_4	teams.players.player, date, teams.name	3/6	1.364
	q_5	teams.players.player, teams.name, teams.home,	2/6	1.157
DBLP	q_6	journal, year	0/2	3.616
	q_7	year, type	0/2	11.843
	q_8	author, year	1/2	36.685
	q_9	type, year, author	2/6	49.993

Table 1: Total time for the querying phase (in seconds)

5.2 OLAP Enabling

For OLAP enabling, we measure the time to check each single AFD and the time to retrieve the cardinality of each level. Tables 2 and 3 show, for each AFD checked between a pair of levels, the number of processed documents, the AFD strength, and the checking time. The difference in the number of documents (or the data size) for a given dataset is to due to the flattening operation that is performed when executing Q_{AFD}. The results clearly show that the querying time t_{AFD} depends on the levels and their depths. This is apparent for the AFDs involving the author attribute in Table 3, which have a higher execution time. This is due to 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.

[2]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.

$l \rightsquigarrow l'$	#Docs	$strength(l, l')$	t_{AFD}
id ⇝ date	32K	**100%**	0.113
date ⇝ id		13.96%	0.065
t.name ⇝ id	64K	0.05%	0.181
t.name ⇝ date		0.05%	0.177
t.name ⇝ t.abbreviation		**100%**	0.175
t.name ⇝ t.home		**50%**	0.168
t.abbreviation ⇝ id	64K	0.05%	0.169
t.abbreviation ⇝ date		0.05%	0.169
t.abbreviation ⇝ t.name		**100%**	0.161
t.abbreviation ⇝ t.home		**50%**	0.163
t.home ⇝ id	64K	0.01%	0.169
t.home ⇝ date		0.02%	0.159
t.home ⇝ t.name		2.77%	0.179
t.home ⇝ t.abbreviation		2.77%	0.169
t.players.player ⇝ id	644K	0.34%	1.285
t.players.player ⇝ date		0.34%	1.278
t.players.player ⇝ t.name		**34.74%**	1.113
t.players.player ⇝ t.abbrev.		**34.74%**	1.087
t.players.player ⇝ t.home		**50.54%**	1.054

Table 2: Time for AFD checks on the Games dataset (in seconds)

$l \rightsquigarrow l'$	#Docs	$strength(l, l')$	t_{AFD}
year ⇝ type	5.4M	**21.76%**	10.945
year ⇝ journal		0.35%	3.373
year ⇝ booktitle		0.16%	4.461
type ⇝ year	5.4M	2.07%	11.211
type ⇝ journal		0.12%	8.181
type ⇝ booktitle		0.03%	4.824
journal ⇝ year	5.4M	**7.09%**	3.365
journal ⇝ type		**99.94%**	8.166
journal ⇝ booktitle		no relationship	1.841
booktitle ⇝ year	5.4M	**29.02%**	4.517
booktitle ⇝ type		**54.97%**	4.949
booktitle ⇝ journal		no relationship	1.852
author ⇝ type	12M	**42.98%**	43.231
author ⇝ year		**36.13%**	36.685
author ⇝ journal		**42.31%**	15.401
author ⇝ booktitle		**31.92%**	18.881

Table 3: Time for AFD checks on the DBLP dataset (in seconds)

Dataset	l	$\|l\|$	t_{card}
Games	id	31686	0.124
	date	4424	0.019
	teams.name	36	0.057
	teams.abbreviation	36	0.056
	teams.home	2	0.049
	teams.players.player	2191	0.622
	All levels	—	0.927
DBLP	author	1859242	12.298
	journal	1632	0.011
	booktitle	10811	0.063
	year	82	0.001
	type	8	0.001
	All levels	—	12.374

Table 4: Time for cardinality checks (in seconds)

Table 4 shows for each level its cardinality and the time to retrieve it. From these results, we can conclude that the efficiency of checking cardinalities also depends on the levels. t_{card} is normally small, since MongoDB can use indexes instead of collection scans. The high value of t_{card} for the level teams.players.player is due to the non-use of nested attribute indexes to answer the corresponding query. As to level author, Q_{card} failed to return the result because the BSON document size (16MB) was exceeded. So we had to define a different query that could not benefit from indexes and suffered from the memory limit discussed earlier.

Using cardinalities to reduce the search space significantly improves the overall performance, since checking a cardinality is faster than checking an AFD. Besides, the number of feasible AFDs when using cardinalities (8, i.e., their strength is shown in bold in Table 2) is less than half the total number of all checked AFDs (19, i.e., the number of rows in Table 2). The same applies for the DBLP dataset, where only 10 of 16 AFDs are feasible.

Finally, Table 5 shows the overall performance of the OLAP enabling phase for our nine md-queries (for all levels in the query): the number of roll-up and drill-down relationships discovered, the number of checks avoided in roll-up and drill-down discovery, and the total time spent. The latter (t_{OLAP}) is calculated as the sum of the times for checking each AFD, plus the time for checking all cardinalities. The results show that our approach for discovering hierarchies effectively reduces the number of AFD checks by using cardinalities and by relying on the dataset structure. Furthermore, the time required by OLAP enabling is reasonable for all md-queries (2 minutes at most), which proves that our performances fit real-time contexts.

Dataset	q	#Roll-up, #Drill-down	#Avoided	t_{OLAP}
Games	q_1	1,0	5/8, 6/8	2.657
	q_2	0,1	6/8 , 5/8	2.657
	q_3	1,0	7/9, 5/9	3.706
	q_4	0,1	5/9, 7/9	3.685
	q_5	0,0	7/9, 7/9	2.513
DBLP	q_6	1,0	4/6 , 2/6	89.940
	q_7	0,1	6/6 , 0/6	113.287
	q_8	0,0	3/6 , 3/6	107.401
	q_9	0,1	4/6 , 2/6	67.653

Table 5: Total time for the OLAP enabling phase (in seconds)

6. CONCLUSION

In this paper we have proposed an interactive schema-on-read approach for enabling OLAP on document stores. To this end, AFDs are mined and used to automate the discovery of multidimensional structures. The user interaction is limited to the selection of multidimensional concepts and to a proper choice of the aggregation operators. After validating user queries from the multidimensional point of view and refining the underlying multidimensional schema, we adopt a smart strategy to efficiently build local portions of 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 evolution of data and schemata in a collection must be considered; we intend to address this issues by searching for temporal FDs.

7. REFERENCES

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

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

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

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

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

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

[7] M. Golfarelli, S. Graziani, and S. Rizzi. Starry vault: Automating multidimensional modeling from data vaults. In *Proc. ADBIS*, pages 137–151, 2016.

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

[9] M. Golfarelli, S. Rizzi, and B. Vrdoljak. Data warehouse design from XML sources. In *Proc. DOLAP*, pages 40–47, 2001.

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

[11] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *Proc. of SIGMOD*, pages 647–658, 2004.

[12] C. S. Jensen, T. Holmgren, and T. B. Pedersen. Discovering multidimensional structure in relational data. In *Proc. DaWaK*, pages 138–148, 2004.

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

[14] M. Klettke, U. Störl, and S. Scherzinger. Schema extraction and structural outlier detection for JSON-based NoSQL data stores. In *Proc. BTW*, pages 425–444, 2015.

[15] H. J. Lenz and A. Shoshani. Summarizability in OLAP and statistical databases. In *Proc. SSDBM*, pages 132–143, 1997.

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

[17] Z. H. Liu, B. Hammerschmidt, D. McMahon, Y. Lu, and H. J. Chang. Closing the functional and performance gap between SQL and NoSQL. In *Proc. SIGMOD*, pages 227–238, 2016.

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

[19] C. Phipps and K. C. Davis. Automating data warehouse conceptual schema design and evaluation. In *Proc. DMDW*, pages 23–32, 2002.

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

[21] S. Rizzi, E. Gallinucci, M. Golfarelli, O. Romero, and A. Abelló. Towards exploratory OLAP on linked data. In *Proc. SEBD*, pages 86–93, 2016.

[22] O. Romero and A. Abelló. Multidimensional design by examples. In *Proc. DaWaK*, pages 85–94, 2006.

[23] O. Romero and A. Abelló. Automating multidimensional design from ontologies. In *Proc. DOLAP*, pages 1–8, 2007.

[24] O. Romero, D. Calvanese, A. Abelló, and M. Rodríguez-Muro. Discovering functional dependencies for multidimensional design. In *Proc. DOLAP*, pages 1–8, 2009.

[25] D. S. Ruiz, S. F. Morales, and Jesus Garcia Molina. Inferring versioned schemas from NoSQL databases and its applications. In *Proc. ER*, pages 467–480, 2015.

[26] K. Valeri. https://thecodebarbarian.wordpress.com, 2014.

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

[28] B. Vrdoljak, M. Banek, and S. Rizzi. Designing web warehouses from XML schemas. In *Proc. DaWaK*, pages 89–98, 2003.

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