Approximate OLAP of Document-Oriented Databases: a Variety-Aware Approach

Approximate OLAP of Document-Oriented Databases: a Variety-Aware Approach[☆]

Enrico Gallinucci, Matteo Golfarelli, Stefano Rizzi[*]

DISI – University of Bologna, Viale Risorgimento 2, 40136 Bologna, Italy

Abstract

Schemaless databases, and document-oriented databases in particular, are preferred to relational ones for storing heterogeneous data with variable schemas and structural forms. However, the absence of a unique schema adds complexity to analytical applications, in which a single analysis often involves large sets of data with different schemas. In this paper we propose an original approach to OLAP on collections stored in document-oriented databases. The basic idea is to stop fighting against schema variety and welcome it as an inherent source of information wealth in schemaless sources. Our approach builds on four stages: schema extraction, schema integration, FD enrichment, and querying; these stages are discussed in detail in the paper. To make users aware of the impact of schema variety, we propose a set of indicators inspired by the definition of attribute density. Finally, we experimentally evaluate our approach in terms of efficiency and effectiveness.

Keywords: NoSQL, Document-Oriented Databases, Multidimensional Modeling, OLAP

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

[*]Corresponding author

Email addresses: enrico.gallinucci@unibo.it (Enrico Gallinucci), matteo.golfarelli@unibo.it (Matteo Golfarelli), stefano.rizzi@unibo.it (Stefano Rizzi)

1. Introduction

Recent years have witnessed an erosion of the relational DBMS predominance to the benefit of DBMSs based on alternative representation models (e.g., document-oriented and graph-based) which adopt a *schemaless* representation for data. Schemaless databases are preferred to relational ones for storing heterogeneous data with variable schemas and structural forms; typical schema variants within a collection consist in missing or additional fields, in different names or types for an field, and in different structures for instances [1]. The absence of a unique schema grants flexibility to operational applications but adds complexity to analytical applications, in which a single analysis often involves large sets of data with different schemas. Dealing with this complexity while adopting a classical data warehouse design approach would require a notable effort to understand the rules that drove the use of alternative schemas, plus an integration activity to identify a common schema to be adopted for analysis — which is quite hard when no documentation is available. Furthermore, since new schema variations are often made, a continuous evolution of both ETL process and cube schemas would be needed.

In this paper we propose an original approach to multidimensional querying and OLAP on schemaless sources, in particular on collections stored in document-oriented databases (DODs) such as MongoDB[1]. The basic idea is to stop fighting against data heterogeneity and schema variety, and welcome it as an inherent source of information wealth in schemaless sources. So, instead of trying to hide this variety, we show it to users (basically, data scientist and data enthusiasts) making them aware of its impact. Specifically, the distinguishing features of our approach are as follows.

- To the best of our knowledge, this is the first approach to propose a form of approximated OLAP analyses on document-oriented databases

[1]The concept of *document* in the context of DODs should not be confused with the one considered in Information Retrieval, where a document is basically a collection of keywords and is organized into chapters, sections, paragraphs, etc.

that embraces and exploits the inherent variety of documents.

- Multidimensional querying and OLAP are carried out directly on the data source, without materializing any cube or data warehouse.
- We adopt an *inclusive* solution to integration, i.e., the user can include a concept in a query even if it is present in a subset of documents only. We cover both inter-schema and intra-schema variety, specifically we cope with missing fields, different levels of detail in instances, different field naming.
- Our approach to reformulation of multidimensional queries on heterogeneous documents grounds on a formal approach [2], which ensures its correctness and completeness.
- We propose a set of indicators to make the user aware of the quality of the query result.

This paper extends our previous contribution [3], mainly by providing the full formalization of the approach, a deeper discussion of the query indicators, and an extensive experimental evaluation.

Remarkably, this is not yet another paper on multidimensional modeling from non-traditional data sources. Indeed, our goal is not to design a single "sharp" schema where source fields are either included or absent, but rather to enable an OLAP experience on some sort of "soft" schema where each source field is present to some extent.

The paper outline is as follows. After giving an overview of our approach in Section 2, in Sections 3, 4, 5, and 6 we describe its four stages, namely, schema extraction, schema integration, FD enrichment, and querying. Then, in Section 7 we provide a performance evaluation of the developed prototype, while the related literature is discussed in Section 8. Finally, in Section 9 we draw the conclusions.

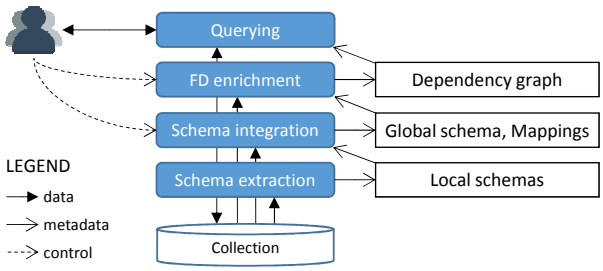

Figure 1: Approach overview

2. Approach overview

Figure 1 gives an overview of the approach: in blue the different stages of the approach, on the right the metadata produced/consumed by each stage. Remarkably, all schema-related concepts are stored as metadata, so no transformation has to be done on source data. User interaction is required at most stages. Although the picture suggests a sequential execution of the stages, it simply outlines the ordering for the first iteration. In the scenario that we envision, the user starts by analyzing the first results provided by the system, then iteratively injects additional knowledge into the different stages to refine the metadata and improve the querying effectiveness. We now provide a short description of each stage; a deeper discussion will be provided in the following sections.

Schema extraction (Section 3). The goal of this stage is to identify the set of distinct *local schemas* that occur inside a collection of documents. To this end we provide a tree-like definition for schemas which models arrays by considering the union of the schemas of their elements. This is a completely automatic stage which requires no interaction with the user.

Schema integration (Section 4). At this stage we rely on inter-schema mappings and schema integration techniques to determine a (tree-like) *global schema* that gives the user a single and comprehensive description of the contents of the collection. In principle, this stage could be completely automated. In practice, the best results can be obtained through a semi-automatic ap-

proach, that allows users to manually validate/refine the mappings proposed by the system.

FD enrichment (Section 5). Traditional OLAP analyses are carried out on multidimensional cubes. To enable the OLAP experience in our setting, a multidimensional representation of the collection must be derived from the global schema. In particular, we introduce the notion of *dependency graph*, i.e., a graph that provides a multidimensional view of the global schema in terms of the functional dependencies (FDs) between its fields. Some FDs can be inferred from the structure of the schema, others by analyzing data; given the expected schema variety, we specifically look for approximate FDs.

Querying (Section 6). The last stage consists in delivering the OLAP experience to the user by enabling the formulation of multidimensional queries on the dependency graph and their execution on the collection. First of all, each formulated query is validated against the requirements of well-formedness proposed in the literature [4]. Then, the query is translated to the query language of the DOD and reformulated into multiple queries, one for each local schema in the collection; the results presented to the user are obtained by merging the results of the single local queries. To make the user aware of the impact of schema variety, we show her a set of indicators describing the quality and reliability of the query result.

The motivation example that we use across the paper is based on a real-world collection of workout sessions, obtained from a worldwide company selling fitness equipment. Figure 2 shows a sample document in the collection, organized according to three nesting levels:

1. The first level contains information about the user, including the facility in which the session took place, the date, and the total duration in minutes.
2. The Exercises array contains an object for every exercise carried out during the session, with information on the type of exercise, and the total calories.
3. The Sets array contains an object for every set that the exercise was split into. For example, the "leg press" exercise has been done in multiple sets,

```
[   {   "_id" : ObjectId("54a4332f44cfc02424f961d4"),
        "User" :
        {   "FullName" : "John Smith",
            "Age" : 42  },
        "StartedOn" : ISODate("2017-06-15T10:20:44.000Z"),
        "Facility" :
        {   "Name" : "PureGym Piccadilly",
            "Chain" : "PureGym"  },
        "SessionType" : "RunningProgram",
        "DurationMins": 90,
        "Exercises" :
        [   {   "Type" : "Leg press",
                "ExCalories" : 28,
                "Sets" :
                [   {   "Reps" : 14,
                        "Weight" : 60  },
                    . . .
                ] },
            {   "Type" : "Tapis roulant" },
            . . .
        ]
    } ,
    . . .
]
```


Figure 2: An excerpt of the WorkoutSession collection

the first of which comprises 14 repetitions with a weight of 60 kilograms, for a total of 28 calories.

3. Schema extraction

The goal of this stage is to introduce a notion of (local) schema for a document, to be used in the integration stage to determine a (global) schema for a collection and then, in the FD enrichment stage, to derive an OLAP-compliant representation of the collection itself.

The notion of a *document* is the central concept of a DOD, and it encapsulates and encodes its data in some standard format. The most widely adopted format is currently JSON, which we will use as a reference in this work.

Definition 1 (Document and Collection). *A* document *d is a JSON object. An* object *is formed by a set of key/value pairs (aka* fields*); a* key *is string, while a* value *can be either a primitive value (i.e, a number, a string, or a Boolean), an array of values, an object, or* null*. A collection D is an array of documents.*

Example 1. *Figure 2 shows a document excerpted from the* WorkoutSession *collection; it contains numbers (e.g.,* Age*), strings (e.g.,* Chain*), objects (e.g.,* User*), and arrays (e.g.,* Exercises*). Conceptually, a session is done by a user at a facility; it includes a list of exercises, each possibly comprising several sets.* □

Since there is no explicit representation of schemas in documents, multiple definitions of schema are possible for the schemas of collections and documents —with different levels of conciseness and precision. The main difference in these definitions lies in how they cope with *inter-document* variety and *intra-document* variety.

- Inter-document variety impacts on the definition of the schema for a collection, as it concerns the presence of documents with different fields. This issue is usually dealt with in one of two ways: either by defining the schema of the collection as the union/intersection [5, 6] of the most frequent fields, or by keeping track of every different schema [7]. Our work mixes the above mentioned approaches in that it builds a global schema starting from local schemas.

- Intra-document variety impacts on the definition of the schema for a document, and is mainly related to the presence in a document of a heterogeneous array. For instance, an array of objects can mix objects with different fields (e.g., the first objects of the Exercises array in Figure 2 contains fields that are missing from the second one). In this work we adopt a simple representation that, like in [5, 8], considers the union of the values contained in the array.

We start by giving a "structural" definition of a schema as a tree, then we reuse it to define the schema of a document and, in Section 4, the schema of a collection.

Definition 2 (Schema). *A* schema *is a directed tree $s = (F, A)$ with root $r \in F$, where F is a set of fields and A is a set of arcs representing the relationships between arrays and the contained fields. In particular,*

1. *$F = F^{arr} \cup F^{prim}$, F^{arr} is a set of array fields (including r), and F^{prim} is a set of primitive fields;*

2. *A includes arcs from fields in F^{arr} to fields in $F^{arr} \cup F^{prim}$.*

Each field $f \in F$ has a name, $key(f)$, a unique pathname (obtained by concatenating the names of the fields along the path from r to f, with the exclusion of r), and a type, $type(f)$ ($type(f) \in \{\mathsf{number}, \mathsf{string}, \mathsf{Boolean}\}$ for all $f \in F^{prim}$, $type(f) = \mathsf{array}$ for all $f \in F^{arr}$). Given field $f \neq r$, we denote with $arr(f)$ the array $a \in F^{arr}$ such that $(a, f) \in A$. Also, given two arrays a_i and a_j we say that $a_i \triangleright a_j$ if a_i is nested in a_j.

To define the schema of a specific document we need to add identifiers to arrays. We denote with $id(a)$ the primitive field that *identifies* an object within array a. Documents always contain an identifier, $id(r) = \_\mathsf{id}$. Conversely, array objects may not contain such a field, but still they can be univocally identified by their positional index within the array. Therefore, given array a, $id(a)$ can be recursively defined as the concatenation of $id(arr(a))$ and the positional index within a; it is $key(id(a)) = \_\mathsf{id}$ and $type(id(a)) = \mathsf{string}$.

Definition 3 (Schema of a Document). *Given document $d \in D$, the* schema of d *is the schema $s(d) = (F^{arr} \cup F^{prim}, A)$ such that*

1. *F^{arr} includes a field for each array in d, labelled with the corresponding key and type, plus a root r labelled with the name of D.*

2. *F^{prim} includes (i) a field for each primitive in d, and (ii) a field for each $id(a)$ with $a \in F^{arr}, f \neq r$; every field is labelled with its corresponding key and type (keys of primitives within an object field are "flattened", i.e., prefixed with the object's key);*

3. *A includes (i) an arc (r, f) for each field f such that $key(f)$ appears as a key in the root level of d, and (ii) an arc (a, f) iff $key(f)$ appears as a key in an object of array a.*

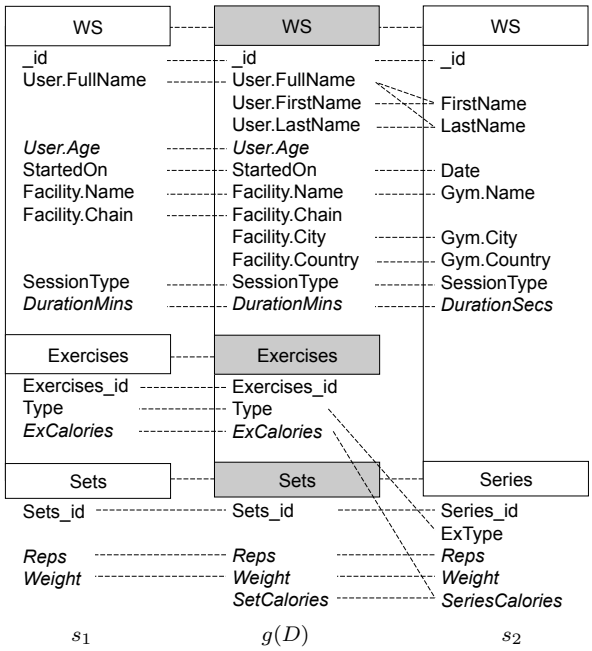

Figure 3: The schema of the JSON document in Figure 2 (s_1), another schema of the same collection WS (s_2), and the global schema ($g(D)$)

Example 2. *Figure 3 shows the schema s_1 of the document represented in Figure 2, part of the* WorkoutSession *collection (from now on, abbreviated in* WS*). Each array is represented as a box, with its child primitives listed below (numeric primitives are in italics). Object fields are prefixed with the object key (e.g.,* Facility.Chain*). The vertical lines between boxes represent inter-array arcs, with the root* WS *on top. It is arr*(Exercises.Type) = Exercises *and id*(Exercises) = Exercises._id*, while* Exercises.Sets ▷ Exercises ▷ WS*.* □

Given collection D, we denote with $S(D)$ the set of distinct schemas of the documents in D (where two fields in the schemas of two documents are considered equal if they have the same pathname).

$$S(D) = \bigcup_{d \in D} s(d)$$

Given $s \in S(D)$, we denote with D^s the set of documents in D such that

$s(d) = s$.

4. Schema integration

The goal of this stage is to integrate the distinct, *local* schemas extracted from D to obtain a single and comprehensive view of the collection, i.e., a *global schema*, and its mappings with each local schema.

4.1. Mappings

A mapping is defined as follows:

Definition 4 (Mapping). *Given two schemas s_i and s_j, a* mapping *from s_i to s_j can be either*

- *an* array mapping *with form $\langle a, a' \rangle$, where $a \in F_i^{arr}$ and $a' \in F_j^{arr}$;*
- *a* primitive mapping *with form $\langle P, P', \phi \rangle$, where $P \subseteq F_i^{prim}$, $P' \subseteq F_j^{prim}$, and ϕ is a* transcoding function*, $\phi : Dom(P) \rightarrow Dom(P')$.*

We emphasize that this definition supports many-to-many mappings between primitive fields.

The definition of the global schema for a collection is based on the mappings determined.

Definition 5 (Global Schema). *Given collection D and the corresponding set of schemas $S(D) = \{s_1, \ldots, s_n\}$, the* global schema *of D is a schema $g(D) = (F, A)$ where*

1. *for every $s_i \in S(D)$ there is a mapping $\langle r_i, r \rangle$ from the root of s_i to the root of $g(D)$;*
2. *every field f in each s_i is involved in at least one mapping to the fields of $g(D)$;*
3. *every field f in $g(D)$ is involved in at least one mapping from some s_i;*

4. *A includes an arc* (f, f') *only if there is an arc* (f_i, f_i') *in a schema* s_i *such that there is a mapping from* f_i *to* f *and one from* f_i' *to* f'.

Note that, since $g(D)$ is a tree, it must include exactly one path from the root to each other node, which excludes "dangling" fields and transitive arcs.

Example 3. *Figure 3 shows two sample schemas from the* WS *collection (*s_1 *and* s_2*) and the corresponding global schema* $g(D)$*; mappings are represented with dotted lines. An example of array mapping from* s_2 *to global schema* $g(D)$ *is*

$$\langle \mathsf{Series}, \mathsf{Exercises.Sets} \rangle$$

Examples of primitive mappings are

$$\langle \{\mathsf{Date}\}, \{\mathsf{StartedOn}\}, \phi_1 \rangle$$
$$\langle \{\mathsf{FirstName}, \mathsf{LastName}\}, \{\mathsf{User.FullName}\}, \phi_2 \rangle$$
$$\langle \{\mathsf{Series.ExType}\}, \{\mathsf{Exercise.Type}\}, \phi_1 \rangle$$

where ϕ_1 *is the identity function while* ϕ_2 *is a function that concatenates two strings.* □

A transcoding function transforms values of a set of fields into values of another set of fields; it is needed for each primitive mapping to enable query reformulation in presence of selection predicates as well as to enable the results obtained from all documents to be integrated (see Section 6.2). On the other hand, array mappings are not associated to a transcoding function because arrays are just containers and do not have values themselves.

Due to the already mentioned inter-document variety, a field f of the global schema may not be available in every local schema (e.g., Facility.Chain is absent in s_2 in Figure 3); therefore we need a measure of the *support* of f with respect to the different schemas in collection D. Intuitively, given the nested structure of documents, the support of f could be defined as the percentage of times that f occurs among the objects of $arr(f)$. However, due to the fact that f may

occur at different depths in different documents (e.g., if $f = \mathsf{Exercises.ExCalories}$ in the global schema of Figure 3, $arr(f)$ is Exercises in s_1 and Exercises.Sets in s_2), this measure must be computed locally to each schema and then aggregated to get a global measure. Thus, we define the *global support* of f as the weighted average of the *local support*s calculated on the distinct schemas.

Definition 6 (Local Support of a Field). *Given a document schema $s = (F, A)$, the local support of a field $f \in F$ is recursively defined as:*

$$locSupp(f, s) = \begin{cases} 1, \ \textit{if } f \equiv r \\ \sum_{s \in D^s} perc(f) \cdot locSupp(arr(f), s), \ \textit{otherwise} \end{cases}$$

where $perc(f)$ is the percentage of objects of $arr(f)$ which include f.

Note that the support of f is weighted on the support of its array $arr(f)$; this is because, for instance, f may occur in every object of $arr(f)$ but $arr(f)$ may be missing for some object of $arr(arr(f))$. As a result, it is always $locSupp(f, s) \leq locSupp(arr(f), s)$.

Definition 7 (Global Support of a Field). *Given collection D and the set of distinct schemas $S(D)$, the global support of a field $f \in F$ is:*

$$gloSupp(f) = \sum_{s \in S(D)} locSupp(f, s) \cdot \frac{|D^s|}{|D|}$$

where $|D^s|$ is the number of documents with schema s and $|D|$ is the overall number of documents.

Example 4. *In our working example, let the* WS *collection have 100 documents (i.e., $|D| = 100$) evenly distributed between s_1 and s_2 (i.e., $|D^{s_1}| = |D^{s_2}| = 50$). Let $f =$* Facility.Name *occur 40 times in s_1 and 20 times in s_2; then, $locSupp(f, s_1) = \frac{40}{50} * 1 = 0.8$, $locSupp(f, s_2) = \frac{20}{50} * 1 = 0.4$ and $gloSupp(f) = 0.8 * 0.5 + 0.4 * 0.5 = 0.6$.* □

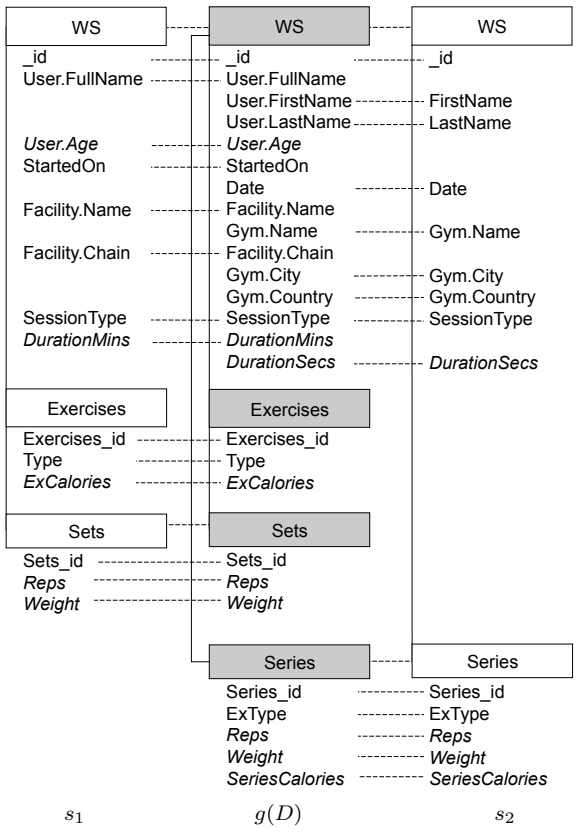

Figure 4: The schema of the JSON document in Figure 2 (s_1), another schema of the same collection WS (s_2), and the preliminary global schema ($g(D)$)

4.2. Integration method

Several techniques have been proposed in the literature for the integration of different schemas [9], which could be reused in our scenario. For instance, the adoption of a *ladder* integration strategy would allow to build the global schema by (i) taking one local schema as the global schema; (ii) iteratively taking each other local schema, finding its mappings onto the global schema, and updating the global schema accordingly. However, some mappings may be missed by adopting a purely incremental strategy (unless a second iteration on the local schemas is done).

The approach we follow is made of two steps. The first step is automatic and consists in defining a preliminary global schema as the simple name-based union

of all local schemas, according to a *one-shot* strategy [9] and similarly to [10]. This gives users an overall understanding of the schemas in the collection and allows to effortlessly create a first set of trivial mappings. Consistently with Section 3, to determine the union we consider that fields in different local schemas are equal if they share the same pathname. Thus, we start from a global schema $g(D) = (F, A)$ where F is the union of the fields for every $s \in S(D)$; arcs and mappings are defined accordingly as of Definition 5, with all transcodings for primitive mappings set to the identity function. Figure 4 shows the preliminary global schema for our working example; here, for instance, primitive field SessionType has the same pathname in both local schemas s_1 and s_2, so it appears only once in the global schema.

In the second step, the preliminary global schema is refined by merging matching (sets of) fields in the global schema. Existing tools (e.g., Coma 3.0 [11]) can be used to automatically find a list of possible matches between arrays and primitives (together with a measure of similarity) and show them to the user; then she will decide which of those should be kept. This way, the user can also define additional mappings that the automated procedure failed to find. To efficiently provide the list of possible matches, we restrict the search space by evaluating a match between two fields only if they are mutually exclusive within the documents. Specifically, the match between two array fields $a, a' \in g(D)$ is evaluated only if there is no local schema in $S(D)$ where a and a' coexist; the same for two sets of primitive fields P and P'. The intuition behind this choice is that, if two fields appear together in the same local schema, then they most probably express two different kinds of information.

Merging two matching sets of primitive fields P and P' in $g(D)$ requires some mappings to be replaced. For simplicity, consider singleton sets: $P = \{f\}$, $P' = \{f'\}$. Only one of the two matching fields, say f, is kept in $g(D)$; f' is dropped from $g(D)$ and all mappings $\langle \{f'_i\}, f', \phi' \rangle$ (where f'_i is a field in local schema s_i) are replaced with $\langle \{f'_i\}, f, \phi'' \rangle$. Similarly for non-singleton sets. For instance, consider FirstName and LastName, which the user matches to FullName; this requires dropping both FirstName and LastName and replacing mappings

$\langle\{\mathsf{FirstName}\}, \{\mathsf{FirstName}\}, \phi_1\rangle$ and $\langle\{\mathsf{LastName}\}, \{\mathsf{LastName}\}, \phi_1\rangle$ with a single mapping $\langle\{\mathsf{FirstName}, \mathsf{LastName}\}, \{\mathsf{FullName}\}, \phi_2\rangle$. Similarly, merging two array fields recursively impacts the children of the array that is removed from the global schema.

5. FD enrichment

The goal of this stage is to propose a multidimensional view of the global schema to enable OLAP analyses. The main informative gap to be filled to this end is the identification of hierarchies, which in turn relies on the identification of FDs between fields in the global schema.

While in relational databases FDs are represented at the schema level by means of primary and referential integrity constraints, the same is not true in DODs. Yet, identifiers are present in DODs: each collection has its (explicit) _id field and, as discussed in Section 3, every nested object has its own (implicit) identifier (i.e., $id(a)$ with $a \in F^{arr}$). The presence of these identifiers implies the existence of some exact FDs, that we call *intensional* as they can be derived from the global schema without looking at the data. In particular, given global schema $g(D) = (F^{arr} \cup F^{prim}, A)$ and array $a \in F^{arr}$, we can infer that:

- $id(a) \rightarrow f$ for every $f \in F^{prim}$ such that $arr(f) = a$, i.e., the identifier of a determines the value of every primitive in a (e.g., $\mathsf{\_id} \rightarrow \mathsf{SessionType}$ and $\mathsf{Exercises.Exercises\_id} \rightarrow \mathsf{Exercises.Type}$);
- if $a \neq r$, then $id(a) \rightarrow id(arr(a))$ —i.e., the identifier of a determines the identifier of $arr(a)$ (e.g., $\mathsf{Exercises.Exercises\_id} \rightarrow \mathsf{\_id}$ and $\mathsf{Exercises.Sets.Sets\_id} \rightarrow \mathsf{Exercises.Exercises\_id}$); this is trivial, since $id(arr(a))$ is part of $id(a)$.

In practice, additional FDs can exist between primitive nodes, though they cannot be inferred from the schema; so, they can only be found by querying the data. More precisely, since DODs may contain incomplete and faulty data, we have to look for *approximate FDs* (AFDs [12]), i.e., FDs that "mostly" hold on data —as done for instance in [13, 14, 15].

Definition 8 (Approximate Functional Dependency). *Given two fields f and f', we say AFD $\gamma = f \rightsquigarrow f'$ holds if $1 - acc(\gamma) \leq \epsilon$, where $acc(\gamma) = \frac{|f|}{|(f,f')|}$ denotes the ratio between the number of unique values of f and the number of unique values of (f, f'), and ϵ is a user-defined tolerance. It is $acc(\gamma) \in (0..1]$.*

As a practical guideline to set the tolerance ϵ, we suggest that it is not kept too strict. Of course a loose tolerance will lead to detecting several AFDs, some of which will probably be actually not true in the application domain. However, later in this section, we will explain how AFD detection must be followed by a user-driven step of editing aimed at removing unwanted AFDs. Besides, our paradigm for approximate OLAP is based on set of indicators (see Section 6.5) that enable users to evaluate in advance the quality of a query before its execution.

Example 5. *Consider fields* Facility.Name *and* Facility.Chain*, with cardinalities* $|\text{Facility.Name}| = 2134$*,* $|\text{Facility.Chain}| = 103$*, and* $|(\text{Facility.Name}, \text{Facility.Chain})| = 2174$*. If* $\epsilon = 0.05$*, it is* $1 - \frac{2134}{2174} = 0.02 \leq \epsilon$*, thus AFD* Facility.Name $\rightsquigarrow$ Facility.Chain *holds.*

Based on Definition 8, checking the existence of AFD $f \rightsquigarrow f'$ requires to compare the cardinalities of f and of (f, f'), which in turns requires two `COUNT DISTINCT` queries to be executed. To detect AFDs, some approaches that were recently devised in the literature (e.g., [14] and [15]) can be reused; interestingly, in [15] the number of queries to be made for AFD detection is effectively reduced thanks to the intensional FDs provided by the global schema.

The procedure we adopt to explore the space of all possible AFDs is sketched in Algorithm 1; it is an adaptation of the well-known *Tane* algorithm [16] and it builds on both [14] and [15] to take advantage of the rules proposed to prune the search space. The goal is to minimize the number of `COUNT DISTINCT` queries to execute, as accessing the data can be quite expensive —even considering that queries must be reformulated from the global schema onto each local schema due to inter-document variety (how this is done is discussed in Section 6.2).

Given the set of primitives F_g^{prim}, the set of candidate AFDs $f \rightsquigarrow f'$, with $f, f' \in F_g^{prim}$, can be represented using an $|F_g^{prim}| \times |F_g^{prim}|$ matrix Z whose rows and columns represent left- and right-hand sides of AFDs, respectively. Given position i in Z, we will denote the corresponding primitive with f_i; thus, $Z[i, j]$ corresponds to f_i and f_j and is set to true if $f_i \rightsquigarrow f_j$ is found to hold on the stored data, to false otherwise.

A naive approach to fill Z would check each single cell (i.e., each possible simple AFD) by querying the collection; actually, most queries can be avoided by orderly exploring the cells of Z, as demonstrated in [14]. Thus, our exploration strategy requires the rows and columns of Z to be ordered by descending cardinality of the corresponding primitive. These cardinalities are calculated in Lines 1–2 by calling the *CountDistinct* procedure (which issues the respective query) on every primitive, while procedure *Initialize* (Line 3) creates the ordered matrix and initializes every cell to NULL. Then, the exploration strategy iterates on the upper diagonal (Lines 4–5) to look for AFDs. The search space is pruned by applying a set of rules; the actual check of AFDs is done in Lines 7–9 and 16–18 by calling *CountDistinct* on a couple of primitives and by checking the inequality of Definition 8. The applied pruning rules are the following:

(a) Given two fields f and f', an exact FD from f to f' cannot hold if $|f| < |f'|$. Thus, if we were looking for exact FDs, we could safely set to false the cells of Z below the diagonal, i.e., those for which $|f| < |f'|$ due to to rows and columns ordering [14]. Conversely, when working with AFDs, we must allow some tolerance on field cardinalities to accomodate possible errors on data. Hence, we can safely avoid to measure $|(f, f')|$ only if $\frac{|f|}{|f'|} < 1 - \epsilon$. This can be easily verified by observing that $|(f, f')| \geq |f'|$, so $\frac{|f|}{|f'|} \geq \frac{|f|}{|(f,f')|}$. But then, $\frac{|f|}{|f'|} < 1-\epsilon$ implies $\frac{|f|}{|(f,f')|} < 1-\epsilon$, so $f \rightsquigarrow f'$ cannot hold. For instance, setting $\epsilon = 0.03$, an AFD from Ex.Type (which has 100 distinct values) to Facility.Chain (103 distinct values) can exist since $\frac{|100|}{|103|} = 0.9708 \geq 1 - 0.03$. *Inverse* AFDs (i.e., cells below the diagonal) are checked in Lines 13–14 and 15–18; notice that, in the first case, the check is inexpensive as it does not

Algorithm 1 AFD detection

Input $g(D)$: a global schema; ϵ: a user-defined tolerance.
Output Z: an AFD matrix

1: **for all** $i \in [1, |F_g^{prim}|]$ **do** ▷ Compute field cardinalities
2: $\quad |f_i| \leftarrow CountDistinct(f_i)$
3: $Z \leftarrow Initialize$
4: **for all** $j \in [2, |F_g^{prim}|]$ **do** ▷ Check the AFDs above the diagonal
5: $\quad$ **for all** $i \in [j-1, 1]$ **do**
6: $\quad\quad$ **if** $(Z[i,j] = NULL) \wedge ((arr(f_i) = arr(f_j) \vee (arr(f_i) \triangleright arr(f_j)))$ **then**
7: $\quad\quad\quad |(f_i, f_j)| \leftarrow CountDistinct(f_i, f_j)$
8: $\quad\quad\quad$ **if** $(1 - \frac{|f_i|}{|(f_i,f_j)|} \leq \epsilon)$ **then**
9: $\quad\quad\quad\quad Z[i,j] \leftarrow TRUE$ ▷ $f_i \rightsquigarrow f_j$ holds
10: $\quad\quad\quad\quad$ **if** $\frac{|f_i|}{|(f_i,f_j)|} = 1$ **then** ▷ Apply transitivity if $f_i \rightarrow f_j$
11: $\quad\quad\quad\quad\quad$ **for all** $k \in [i-1, 1]$ s.t. $Z[k,i] = TRUE$ **do**
12: $\quad\quad\quad\quad\quad\quad Z[k,j] \leftarrow TRUE$ ▷ $f_k \rightsquigarrow f_j$ holds
13: $\quad\quad\quad\quad$ **if** $(1 - \frac{|f_j|}{|(f_i,f_j)|} \leq \epsilon) \wedge ((arr(f_j) = arr(f_i)) \vee (arr(f_j) \triangleright arr(f_i)))$ **then**
14: $\quad\quad\quad\quad\quad Z[j,i] \leftarrow TRUE$ ▷ $f_j \rightsquigarrow f_i$ holds
15: $\quad\quad$ **else if** $(1 - \frac{|f_j|}{|f_i|} \leq \epsilon) \wedge ((arr(f_j) = arr(f_i)) \vee (arr(f_j) \triangleright arr(f_i)))$ **then**
16: $\quad\quad\quad |(f_i, f_j)| \leftarrow CountDistinct(f_i, f_j)$ ▷ $f_i \rightsquigarrow f_j$ has not been checked due to rule (b)
17: $\quad\quad\quad$ **if** $(1 - \frac{|f_j|}{|(f_i,f_j)|} \leq \epsilon)$ **then**
18: $\quad\quad\quad\quad Z[j,i] \leftarrow TRUE$ ▷ $f_j \rightsquigarrow f_i$ holds
19: **return** Z

require any extra call to $CountDistinct$.

(b) AFD $f \rightsquigarrow f'$ may exist iff either $arr(f) = arr(f')$ (i.e., in each document, each value of f is associated to a single value of f') or $arr(f) \triangleright arr(f')$ (i.e., many values of f are associated to a single value of f'). If this is not the case (either because $arr(f') \triangleright arr(f)$ or because there is an array a such that $arr(f) \triangleright a$ and $arr(f') \triangleright a$), then each value of f is associated to many values of f', making it impossible for f to functionally determine f'. For instance, given SessionType (whose array is WS) and Exercises.Type (whose array is Exercises), an AFD can exist only from the latter to the former, not vice versa. This rule is verified in Lines 6 and 13.

(c) Differently from exact FDs, error accumulation due to faulty instances prevents from straightly applying transitivity in AFDs, so knowing that $f \rightsquigarrow f'$ and $f' \rightsquigarrow f''$ does not necessarily imply that $f \rightsquigarrow f''$. Only in case it is $f \rightsquigarrow f'$ and $f' \rightarrow f''$, we can safely infer that $f \rightsquigarrow f''$ (because $|(f, f'')| = |(f, f')|$ since every value in f' references only one value in f''). Conversely, in the general case, though it is possible to derive an upper

bound of $|(f, f'')|$ based on $|f'|$, $|(f, f')|$, and $|(f', f'')|$, this bound is too loose to be really effective in avoiding `COUNT DISTINCT` queries. For this reason, in Lines 10–12 we only consider the case in which $f \to f'$.

Now let Γ be the set of (A)FD found for collection D, which includes both the intensional (exact) FDs inferred from the global schema $g(D)$ as described above and the extensional (approximate) FDs detected by Algorithm 1. Clearly, in the general case it is possible that Γ presents cycles, while hierarchies to be used for OLAP querying are well-known to be directed acyclic graphs (DAGs) [17]. Hence, these cycles must be removed in order to identify hierarchies. This can be done through a user-driven step of editing as in classical data-driven approaches to multidimensional modeling based on FDs (e.g., [13, 18, 14]).

Definition 9 (Dependency Graph). *Given the global schema $g(D) = (F, A)$ and an (acyclic) set of simple and not-trivial (A)FDs Γ, the* dependency graph *is defined by a couple $\mathcal{M} = (F^{prim}, \succ)$ where F^{prim} is the set of primitive nodes in F and $\succ$ is a* roll-up *partial order of F^{prim} derived from Γ. In particular, $f_j \succ f_k$ if either $f_j \rightsquigarrow f_k \in \Gamma$ or $f_j \to f_k \in \Gamma$.*

The differences between a dependency graph and the global schema it is derived from are that

1. the global schema is a tree, the dependency graph is induced by a partial order so it is a DAG;
2. arrays are not present in the dependency graph, but their id's are;
3. arcs express (A)FDs in the dependency graph, syntactical containment in the global schema;
4. differently from the global schema, the dependency graph can include arcs between primitive fields.

Example 6. *Figure 5 shows the dependency graph for our working example. Each primitive field is represented as a circle whose color is representative of*

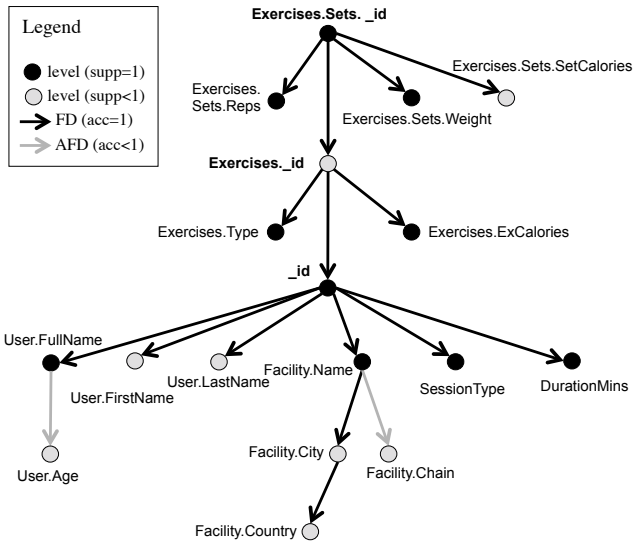

Figure 5: Dependency graph for the global schema in Figure 3

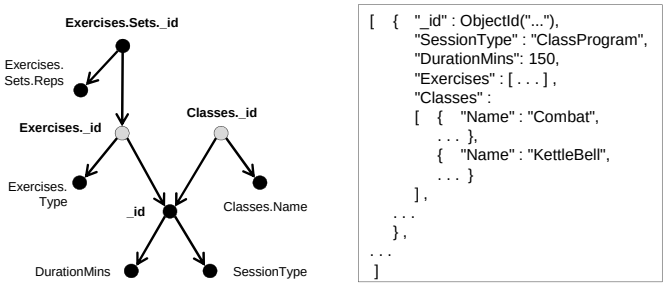

Figure 6: Excerpt of the dependency graph (left) in presence of alternative documents (right)

the field global support (the lighter the tone, the lower the support). Identifiers (e.g., _id*) are shown in bold. Directed arrows are representative of the (A)FDs in* Γ*; for instance, it is* _id $\rightarrow$ Facility.Name *(FDs are shown in black) and* Facility.Name $\rightsquigarrow$ Facility.Chain *(AFDs are shown in grey). Note that, in this case, the dependency graph is a tree, because in the global schema of Figure 3 arrays are nested within each other. A different situation is the one shown in Figure 6, where the collection includes documents with two arrays at the same level, so the dependency graph is not a tree.* □

6. Querying

In this section we describe the final querying stage. We start by providing the definition of a multidimensional query and discussing its correctness (Section 6.1). Then, we discuss the execution of a query, which mainly involves its reformulation from the global schema to the local schemas (Section 6.2) and the translation of each reformulated query into the MongoDB language (Section 6.3). Finally, we introduce a set of indicators describing the quality and reliability of the query result (Section 6.4) and explore the issue of query evolution, i.e., the formulation of new queries by applying OLAP operators on the previous queries (Section 6.5).

6.1. Query formulation

First of all, we define a multidimensional query as follows.

Definition 10 (Md-query). *Given dependency graph* $\mathcal{M} = (F^{prim}, \succ)$*, a* multidimensional query *(from now on, md-query) on* $\mathcal{M}$ *is a triple* $q = \langle G, p, m, \varphi \rangle$ *where:*

- G *is the query* group-by set*, i.e., a non-empty set of fields in* F^{prim} *such that for all couples* f_j, f_k *in* G *it is* $f_j \not\succ f_k$*;*
- p *is an (optional)* selection predicate*; it is a conjunction of Boolean predicates, each involving a field in* F^{prim}*;*
- $m \in F^{prim}$ *is the query* measure*, i.e., the numerical field to be aggregated;*
- φ *is the operator to be used for aggregation (e.g.,* `avg`*,* `sum`*);*
- *there exists in* $\mathcal{M}$ *one single field* $\overline{f}$ *such that either* $\overline{f} \succ f$ *or* $\overline{f} = f$ *for all other fields* f *mentioned in* q *(either in* G*,* p*, or* m*).*

We will refer to all the fields in G and p as the query *levels*. Field $\overline{f}$ is called the *fact* of q (denoted $fact(q)$) and corresponds to the coarsest granularity of $\mathcal{M}$ on which q can be formulated. An example of a case in which a fact cannot be determined is the one in Figure 6, with $G = \{\mathsf{Classes.Name}, \mathsf{Exercises.Type}\}$.

Example 7. *The following md-query on the* WS *collection,* q_1*, measures the average amount of weight lifted by elderly athletes per city and type of exercise:*

$q_1 = \langle$ {Facility.City, Exercises.Type},
(User.Age ≥ 60), Exercises.Sets.Weight, *avg* $\rangle$

It is $fact(q_1)$ = Exercises.Sets._id. □

In [4] the authors outline the constraints that must hold for an md-query to be considered well-formed, namely, the *base integrity constraint* (stating that the levels in the group-by set must be functionally independent on each other) and the *summarization integrity constraint* [19], which in turn 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). Remarkably, Definition 10 already ensures that md-queries meet the base integrity constraint (because the query group-by set cannot include fields related by (A)FDs). As to the summarization integrity constraint, since the goal of our approach is to enable an immediate querying of data with no cleaning beforehand, we adopt a "soft" approach to avoid being too restrictive. So, after each query has been formulated by the user, it undergoes a check (sketched in Algorithm 2) that can possibly return some warnings to inform the user of potentially incorrect results. Specifically, the disjointness constraint ensures that the granularity of the measure is not coarser than the one of the group-by set levels (Line 3); if this is false, the same instance of m will be double counted for multiple instances of the group-by set [4]). The completeness constraint ensures that the levels in the group-by set have full global support (Line 5); this constraint is easily contradicted as it clashes with the schemaless property of DODs. Finally, the compatibility constraint is not considered at all since its verification would require to properly categorize measures (i.e., flow, stock and value-per-unit) and levels (i.e, temporal and non-temporal), but this information can hardly be inferred from the schema or even provided by the user [15].

Algorithm 2 Validity check of an md-query

Input $\mathcal{M} = (F^{prim}, \succ)$: a dependency graph; $q = \langle G, p, m, \varphi \rangle$: an md-query
Output *status*: a validity status
1: $status \leftarrow$ "$valid$"
2: **for each** $f \in G$ **do**
3: **if** $id(arr(f)) \succ id(arr(m))$ **then**
4: $status \leftarrow$ "$warning$" ▷ Disjointness failed
5: **if** $gloSupp(f) < 1$ **then**
6: $status \leftarrow$ "$warning$" ▷ Completeness failed
7: **return** *status*

Example 8. *Query* q_1 *passes the validity check of Algorithm 2 with a completeness warning, because* $gloSupp(\mathsf{Facility.City}) < 1$. *On the other hand,* q_1 *meets the disjointness constraint because*

$$id(arr(\mathsf{Facility.City})) = \mathsf{\_id}$$

$$id(arr(\mathsf{Exercises.Type})) = \mathsf{Exercises.\_id}$$

$$id(arr(\mathsf{Exercises.Sets.Weight})) = \mathsf{Exercises.Sets.\_id}$$

$$\mathsf{Exercises.Sets.\_id} \succ \mathsf{\_id}$$

$$\mathsf{Exercises.Sets.\_id} \succ \mathsf{Exercises.\_id}$$

□

As previously mentioned, an md-query fails the completeness constraint if one or more levels in the group-by set do not have full support. This issue is strictly related to the one of *incomplete hierarchies* in data warehouse design. The related work proposes three alternative strategies to replace missing values in a hierarchy level l_j with placeholders: *balancing by exclusion* (i.e., replacing all missing values with a single value "Other"), *downward balancing* (replacing with values from the closest level l_k such that $l_k \succ l_j$), and *upward balancing* (replacing with values from the closest level l_k such that $l_j \succ l_k$) [18]. Whereas they were originally meant to be applied when populating a data warehouse from an operational source, these strategies can be directly applied at query time, e.g., by using the `$ifNull` operator in MongoDB, which allows to replace a missing value in a field with a custom value or with the value of another field.

Thus, when an md-query fails the completeness constraint, we ask the user to indicate the desired strategy to replace missing values in the levels without full support.

6.2. Query reformulation

Once an md-query has been formulated by the user on the dependency graph corresponding to the global schema, it has to be reformulated on each local schema to effectively cope with inter-document variety. To this end we rely on a formal approach to enable md-query reformulation in a *business intelligence network* (BIN), i.e., a federated data warehouse architecture [2]. The BIN approach presents a framework that enables the reformulation of a query from a source multidimensional schema to a target multidimensional schema and has been proved to be complete and provide all certain answers to the query. In this section, we discuss how the proof of correctness and completeness can be extended to our approach as well. In particular, the reformulation of a query from the global schema to each local schema is necessary in our approach in two situations: (i) when the collection is queried to detect AFDs (Section 5) and (ii) when the user issues an md-query on the collection (Section 6). To this end we need to prove that the data schemas, the mappings, and the queries which we consider in our work are a particular case of those used as a reference in the BIN context.

Data schema. The reference schema in the BIN context is a classical multidimensional schema featuring a fact, a set of hierarchies (each made of levels), and a set of measures (each coupled with an aggregation operator). The dependency graph of Definition 9 can be thought of as a sort of "multi-fact" multidimensional schema with no explicit distinction between levels and measures. However, when an md-query is formulated as in Definition 10, exactly one fact is implicitly determined, group-by levels are explicitly distinguished from measures, and an aggregation operator is coupled to each measure. So, from the data schema point of view, there is no difference between the context of BINs and the one of this paper.

Mappings. The primitive mappings of Definition 4 can be expressed, according to the BIN terminology, using either same or equi-level predicates. same predicates are used for measures, and can be annotated with an expression; since in Definition 10 measures are required to be numerical, the associated transcodings must be translatable into an expression. equi-level predicates are used for levels, and can be directly annotated with a transcoding. Remarkably, in [2] these two types of mappings are called *exact* since they enable non-approximate query reformulations. Note that array mappings are not used for query reformulation but only for determining the global schema, so they are not considered here.

Queries. An md-query (Definition 10) has a group-by set, a (conjunctive) selection predicate, and a measure. A BIN query has a group-by set, a (conjunctive) selection predicate, and an expression involving one or more measures. By simply picking a single measure and the identity expression, situation (ii) is addressed. As to situation (i), i.e., querying aimed at checking AFDs, we remark that the query for checking AFD $l \rightsquigarrow l'$ can be expressed as a BIN query with group-by set $\{l, l'\}$ and a dummy measure, on whose result a simple `COUNT DISTINCT` is then executed.

Based on the considerations above, we can state that a query of the global schema can be correctly reformulated into a set of local queries, one on each local schema. Then, each local query is separately executed on the DOD; specifically, each query must target only the documents that belong to a specific local schema s. This is done in two steps. First, the information about which document has which schema (obtained in the schema extraction stage) is stored in a different collection (called WorkoutSession-schemas in our example) in the following form: a document is created for every schema $s \in S(D)$, containing an array ids with the _id of every document $d \in D^s$. Then, the query on schema s is executed by joining it with the list of identifiers in WorkoutSession-schemas). Finally, a post-processing activity is required to integrate the results coming from the different local queries.

Example 9. *Consider an md-query that calculates the total amount of burnt calories by facility, excluding workout sessions that are shorter than 30 minutes:*

$q = \langle\ \{\mathsf{Facility.Name}\}, (\mathsf{DurationMins} \geq 30), \mathsf{Exercises.ExCalories}, \mathit{sum}\ \rangle$

Consider the local schemas s_1 and s_2 from Figure 3. The reformulation of q onto s_1 has no effect; conversely, the reformulation onto s_2 generates the following md-query:

$q' = \langle\ \{\mathsf{Gym.Name}\}, (\frac{\mathsf{DurationSecs}}{60} \geq 30), \mathsf{Series.SeriesCalories}, \mathit{sum}\ \rangle$

Note that, since `sum` *is a distributive aggregation operator, the aggregation of* Series.SeriesCalories *is correctly computed using only one aggregation. Conversely, the usage of an algebraic operator (e.g.,* `avg`*) would require the computation of pre-aggregates; this is possible by extending Definition 10 to include such computation as discussed in [2].*

6.3. Query execution

In this subsection we explain how, after reformulation, each single query obtained can be translated to MongoDB on the corresponding local schema. Md-queries are translated to MongoDB according to its query language (called *aggregation framework*), which allows to declare a multi-stage pipeline of transformations to be carried out on the documents of a collection. The most important transformations are defined by the following operators:

- `$match`: it is used to apply predicate selections; its equivalent in SQL is the `where` clause.
- `$project`: it is used to apply transformations to the single fields; its equivalent in SQL is the `select` clause.
- `$group`: it is used to group the documents and calculate aggregated values); its equivalent in SQL is the `group by` clause.
- `$unwind`: it is used to unfold an array by creating a different document for every object inside the array); its equivalent in SQL is the `unnest` clause.

Given md-query $q = \langle G, p, m, \varphi \rangle$ on $\mathcal{M}$ and global schema $g(D) = (F, A)$, the translation of q into the MongoDB language is done as follows:

1. As $fact(q)$ represents the granularity level of the query, it is necessary to unfold every array involved in the path from the root r to $fact(q)$. If $fact(q) = r$, no *unwind* stage is necessary; otherwise, an *unwind* stage is added for $fact(q)$ and for every array a in $g(D)$, $a \neq r$, such that $fact(q) \succ id(a)$. The order of these stages reflects the order of the arrays in $g(D)$, beginning from the one closest to r. The process of determining the ordered list of arrays, LA, is described as a recursive function in Algorithm 3.

2. If $p \neq \varnothing$, a `$match` stage is defined listing every selection predicate.

3. A `$project` stage is defined to keep only the fields that are required for the following stages, i.e., m and every group-by level. If there is one (or more) incomplete level $f \in G$ (i.e., such that $gloSupp(f) < 1$), the replacement of the missing values of f is done at this stage, according to the balancing strategy chosen by the user. In case of downward or upward balancing, the closest primitives to f (i.e., any f' such that $f \succ f'$ or $f' \succ f$) may not have full support either; thus, an ordered list of primitives, LP, must be scanned until a primitive with full support is found (or until no more primitives are available). This process is described as a recursive function in Algorithm 4; the process is stopped when $fact(q)$ is encountered (Line 9 in downward balancing) or when the end of the graph is reached (Line 15 in upward balancing). In any case, f may functionally determine (or be determined by) two ore more fields due to a branch in the hierarchy (e.g., Facility.Name $\rightsquigarrow$ Facility.City and Facility.Name $\rightsquigarrow$ Facility.Chain); in this case, the user intervention is required to choose which of the determined field should be chosen for balancing (Lines 8 and 14). Ultimately, a new field named balanced is added and valued $TRUE$ if any of the projected fields has been affected by the balancing strategy, $FALSE$ otherwise.

Algorithm 3 $unfold(a, r)$

Input a: an array field; r: the root
Output LA: the ordered list of array fields to be unfolded

```
1: if a = r then
2:     LA ← ⟨⟩                                          ▷ Empty list
3: else
4:     LA ← ⟨unfold(arr(a), r), a⟩
5: return LA
```

Algorithm 4 $balance(f, dir, q)$

Input f: a primitive field; dir: a balancing direction (either "downward" or "upward"); q: an md-query
Output LP: the ordered list of primitives to be used for balancing f

```
1: if gloSupp(f) = 1 then
2:     LP ← ⟨f⟩
3: else
4:     f_bal ← ∅
5:     if dir = "downward" then
6:         F' ← {f' ∈ F^prim s.t. f' ≻ f}
7:         if |F'| > 1 then
8:             f_bal ← askUser(F')
9:         else if fact(q) ∉ F' then                    ▷ F' = {f'}
10:            f_bal ← f'
11:    else                                             ▷ dir = "upward"
12:        F' ← {f' ∈ F^prim s.t. f ≻ f'}
13:        if |F'| > 1 then
14:            f_bal ← askUser(F')
15:        else if F' ≠ ∅ then                          ▷ F' = {f'}
16:            f_bal ← f'
17:    if f_bal ≠ ∅ then
18:        LP ← ⟨f, balance(f_bal, dir, q)⟩
19:    else
20:        LP ← ⟨f⟩
21: return LP
```

4. A `$group` stage is defined including the fields that identify a group (i.e., every level $f \in G$ plus the balanced field), the measure m to be aggregated, and its aggregation functions φ. Additionally, two new measures named count and count-m are added to count, respectively, the number of aggregated objects and the number of aggregated objects that actually contain a value for m.

The query-independent fields balanced, count, and count-m are needed to calculate the indicators of the query, which will be discussed in Section 6.4.

Example 10. *The MongoDB query obtained from q_1 considering a downward balancing strategy is the following.*

```
db.WS.aggregate({
   { $unwind:  "$Exercises" },
   { $unwind:  "$Exercises.Sets" },
```

```
  { $match: { "User.Age": { $gte: 60 } } },
  { $project: {
    "Facility.City": { $ifNull:
      ["$FacilityCity","$FacilityName"] }
    },
    "Exercises.Type": 1,
    "Exercises.Sets.Weight": 1,
    "balanced": {
      $cond: ["$FacilityCity",false,true]
    }
  } },
  { $group: {
    "_id": {
      "FacilityCity","$FacilityCity",
      "ExercisesType","$Exercises.Type",
      "balanced","$balanced"
    },
    "Exercises.Sets.Weight": {
      $avg: "$Exercises.Sets.Weight"
    },
    "count": { $sum: 1 },
    "count-m": { $sum: {
      $cond: ["$Exercises.Sets.Weight",1,0]
    } }
  } }
}
```

□

The final results shown to the user are composed by further aggregating the results of each query. Note that this operation can be performed in-memory, as OLAP queries usually produce a limited number of records and the transcoding functions provide homogeneous values.

6.4. Query evaluation

In our schemaless scenario, the evaluation of the query results cannot transcend from the evaluation of the query itself. In particular, it is important to understand the coverage of the query with respect to the collection (which may be influenced by the support of the fields, the quality of the mappings, and the

selectivity of the selection predicate), as well as the reliability of the results. For these reason, we introduce some indicators to evaluate the quality of md-queries. Our indicators are inspired by the concept of *attribute density* defined in [20] and adapt it to the OLAP peculiarities.

For the sake of defining the indicators, let n be the number of raw objects that are queried. This value depends not only on the selection predicates, but also on the granularity of the query. For instance, with reference to our working example, if $fact(q) =$ _id (and no selection predicates are provided), then n corresponds to the number of documents in the collection (i.e., $|D|$); otherwise, if $fact(q) =$ Exercises._id, then n corresponds to the total number of exercises, obtained by applying the $unwind operator in the query (as seen in Section 6.3). Also, let E be the set of distinct groups returned by an md-query q; then, we denote with $|e|$ the number of objects that have been aggregated by each group $e \in E$ (measured by the count field as of Section 6.3).

The first indicator we introduce is *level density*: when the group-by set of md-query q includes a level that does not have full support (i.e., in some local schemas that level is null or it does not exist), q fails the completeness constraint (as stated in Section 6.1) and a balancing strategy is adopted to replace the missing values. This leads to a result set in which one or more groups in E contain placeholders. We can then say that $E = E^{orig} \cup E^{bal}$, where E^{orig} contains the groups whose values are the original ones, while E^{bal} contains the groups whose values are determined by balancing. Unless balancing by exclusion is done, identifying the latter groups may not be obvious for users. Thus, on the one hand, we need to distinguish these groups from the others for an easier interpretation of the results; on the other, we want to quantify the weight of these groups with respect to the others. Both evaluations are provided by the level density indicator, which is defined both at group level and at query level. In particular, the *group level density* (i.e., the level density of a specific group e) is:

Table 1: Sample excerpt from the WS collection

_id	SessionType	Facility.Name	Facility.City	Facility.Chain	DurationMins
1	BeginnerProgram	PureGym London	London	null	92
2	BeginnerProgram	PureGym London	London	null	120
3	AdvancedProgram	BestGym London	London	TopOfTheBest	null
4	AdvancedProgram	TopGym	null	TopOfTheBest	108
5	AdvancedProgram	TopGym	null	TopOfTheBest	96

Table 2: Results obtained from Table 1 by grouping on Facility.City and calculating the average DurationMins with downward balancing

Facility.City	avg(DurationMins)
London	106
TopGym	102

$$levelDensity(q, e) = \begin{cases} TRUE, \ \textit{if } e \in E^{orig} \\ FALSE, \ \textit{if } e \in E^{bal} \end{cases}$$

In practice, this indicator is the negation of the balanced field introduced in Section 6.3. Conversely, the *query level density* is calculated as the percentage of aggregated objects belonging to E^{orig}:

$$levelDensity(q) = \frac{\sum_{e \in E^{orig}} |e|}{n}$$

Example 11. *Consider the sample data in Table 1 and md-query* $q = \langle \{\text{Facility.City}\}, TRUE, \text{DurationMins}, \textit{avg} \rangle$*, which determine the results in Table 2: it is* $E^{orig} = \{\textit{London}\}$ *and* $E^{bal} = \{\textit{TopGym}\}$*. Then,* $levelDensity(q, \textit{London}) = TRUE$ *and* $levelDensity(q, \textit{TopGym}) = FALSE$*, while* $levelDensity(q) = 0.60$*.*

The second indicator we introduce is *measure density*. When the measure of the query does not have full support, the values reported in the results offer only a partial summary of the aggregated objects. The more aggregated objects lack the value of that measure, the less the result is precise. Measure density may vary significantly from group to group. For instance, in our example, missing values may be concentrated in facilities that adopt old and faulty equipment;

in this case, though the overall percentage of missing values is significant, the values obtained from the other facilities would be very precise and trustworthy.

Similarly to level density, measure density is evaluated at both the group and query levels. Let $|e|_m$ be the number of objects that have been aggregated by each group $e \in E$ and that actually have a value for m (this is measured by the field count-m as of Section 6.3; clearly, it is $|e|_m \leq |e|$). Then, the *group measure density* of group e is:

$$measureDensity(q, e) = \frac{|e|_m}{|e|}$$

The *query measure density* of query q is:

$$measureDensity(q) = \frac{\sum_{e \in E} |e|_m}{n}$$

Example 12. *Consider the sample data in Table 1 and the same query q as in Example 11. Then, $measureDensity(q, \mathsf{London}) = 0.67$ and $measureDensity(q, \mathsf{TopGym}) = 1.00$, while $measureDensity(q) = 0.80$.*

Level and measure densities enable an evaluation of the reliability of query results. The problem with these indicators is that they must be computed at query time, whereas (considering that the execution time for md-queries is often significant) it would be important for users to have a feedback on the quality of the query *before* it is executed. Although this cannot be done exactly, we can provide an *estimate* of the indicators that, at least, gives a first hint to the user. Level and measure densities essentially depend on the support of the levels and the measure involved in the query, respectively, thus we can rely on the support to provide estimated values at the query level (clearly, indicators at the group-by level cannot be estimated without accessing the data). In particular, we can estimate the measure density of a query as the support of the provided measure:

$$estMeasureDensity(q) = gloSupp(m)$$

The actual measure density of the query varies depending on the query selection predicate. Thus, if no selection predicate is present, the estimate reflects the true value. As to level density, we can define the estimated level density of a query as the product of the supports of the query group-by levels:

$$estLevelDensity(q) = \prod_{l \in G} gloSupp(l)$$

Indeed, the support of a level l can be interpreted as the probability that l is null in a document of the collection. Since there is no a-priori evidence of any inter-level dependency, given two levels l and l', the presence of a null in l or in l' within the same document can be thought of as independent events A and B, so that $P(A \cap B) = P(A) * P(B)$. Nonetheless, the distance of the estimated value from the real one can be significant; in fact, the true level density (in absence of selection predicates) falls within a range that is quite ample:

$$levelDensity(q) \in \left[\max\{0, \sum_{l \in G} gloSupp(l) - (|l| - 1)\}, \min_{l \in G} gloSupp(l) \right]$$

The intuition behind formula 6.4 is that the support of the single group-by levels is an upper bound to the level density of the query. The lower bound depends on how the different levels intersect with each other: two levels do intersect if the sum of their supports is greater than 1, three levels do intersect if the sum of their supports is greater than 2, and so on.

Example 13. *Consider the sample data in Table 1 and md-query* $q = \langle \{$Facility.City, Facility.Chain$\}, TRUE,$ DurationMins, $avg \rangle$. *Then,* $levelDensity(q) \in [max\{0, 0.6 + 0.6 - 1\}, min\{0.6, 0.6\}] = [0.2, 0.6]$, *while* $estLevelDensity(q) = 0.6 * 0.6 = 0.36$ *and* $estMeasureDensity(q) = 0.8$.

A significant improvement in estimating the values of level and measure densities can be achieved by relying on schema profiling techniques. In a previous work [1], schema profiling is introduced as a way to identify the hidden rules that drive the usage of different schemas in a collection. In par-

Algorithm 5 Find local schemas hit by an md-query

Input $p = \{p_1, \ldots, p_k\}$: a selection predicate that conjuncts k simple predicates; R: a set of profiling rules; $S(D)$: the set of local schemas
Output $S_q(D)$: a subset of local schemas

```
1: S_q(D) ← S(D)
2: PredsToCheck ← {p}
3: while PredsToCheck ≠ ∅ do
4:     NextPredsToCheck ← ∅
5:     for all p' ∈ PredsToCheck do
6:         if ∃r ∈ R, r = (p_r, S_r(D)) s.t. p_r = p' then
7:             S_q(D) ← S_q(D) ∩ S_r(D)
8:         else if |p'| > 1 then
9:             for all p_i ∈ p' do
10:                NextPredsToCheck ← NextPredsToCheck ∪ {p' \ {p_i}}
11:    PredsToCheck ← NextPredsToCheck
12: return S_q(D)
```

ticular, a rule indicates that, in presence of a specific value of a field (e.g., SessionType = "AdvancedProgram"), only a specific subset of the local schemas is used. The original goal of schema profiling is to acquire a better knowledge of the collection; however, the rules obtained with schema profiling can be integrated in our approach to refine the estimates of the indicators.

More formally, a *profiling rule* is a couple $r = (p_r, S_r(D))$, where p_r is a selection predicate and $S_r(D) \subseteq S(D)$ is the corresponding subset of local schemas. Given a set of rules R (all with different selection predicates) and an md-query q with selection predicate p defined as the conjunction of k simple predicates, the subset of local schemas hit by q, denoted $S_q(D)$, can be determined as explained in Algorithm 5. The idea is to iterate over R (lines 3 to 11) to find the rule(s) that fit p the most. At first, a rule whose predicate consists of exactly all the k simple predicates of p is searched; if it does not exist, we search for rules with any combination of $(k-1)$ simple predicates, and so on up to single simple predicates. Eventually, $S_q(D)$ is determined as the intersection of all the $S_r(D)$'s for the rules whose predicates (even partially) match p (Line 7).

Once $S_q(D)$ is determined, the estimation of level and measure densities can be improved by restricting the global support of the query levels and measure to the local schemas in $S_q(D)$. This can be achieved by refining the definition

of global support as follows:

$$gloSupp(f) = \sum_{s \in S_q(D)} locSupp(f, s) \cdot \frac{|D^s|}{|D|}$$

Example 14. *Consider the sample data in Table 1 and md-query* $q = \langle \{$Facility.City, Facility.Chain$\}$, (SessionType = "AdvancedProgram"), DurationMins, $avg\rangle$. *As in Example 13 (where* q *had no selection predicate),* $levelDensity \in [0.2, 0.6]$ *and* $estLevelDensity(q) = 0.36$, *while* $estMeasureDensity(q) = 0.8$. *Consider* $S(D) = \{s_a, s_b, s_c\}$, *where* s_a *refers to the first two documents,* s_b *to the third and* $s_c c$ *to the last two. Also, consider rule* $r =$ (SessionType = "AdvancedProgram", $\{s_b, s_c\}$). *Then,* $gloSupp$(Facility.City) *becomes* 0.33 *(instead of* 0.6*),* $gloSupp$(Facility.Chain) *becomes* 1 *(instead of* 0.6*) and* $gloSupp$(DurationMins) *becomes* 0.67 *(instead of* 0.8*). Finally,* $estLevelDensity(q)$ *becomes* $1 * 0.33 = 0.33$ *and* $estMeasureDensity(q)$ *becomes* 0.67 *(which, in this case, are exact estimates).*

6.5. Query evolution

Consistently with an OLAP scenario, an md-query can evolve into another with the application of an OLAP operation; the resulting sequence of queries is called an *OLAP session*. In particular, the permitted operations are the following ones.

- The replacement of the query measure with a different one, or the selection of a different aggregation operator. If a new measure is chosen, a new validity check is required to verify whether the disjointness requirement still holds.
- The addition/removal/modification of a selection predicate. This operation has no impact on the validity of the query.
- The *roll-up* (or *drill-down*) of one of the group-by levels, which leads to replacing a level f with a level f' such that $f \succ f'$ (or $f' \succ f$).

Roll-ups and drill-downs imply a navigation of the dependency graph on the relationships between f and f', which represent (A)FDs. From a multidimensional standpoint, the navigation of an AFD with accuracy lower than 1 leads to a violation of the roll-up semantics, i.e., the results of the second query will not be a correct composition (or decomposition) of the results of the first query. This happens because the FD is not strictly true in some cases, which compromises the correctness of the aggregation. Thus, we evaluate the impact of these operations by means of another indicator. In particular, this indicator quantifies the *accuracy* of the aggregated results of a query during an OLAP session with respect to the results obtained from the previous query. Let ω be the OLAP operation applied to md-query q to obtain q' by rolling up (or drilling down) from level f to f', and let $\Gamma' \subseteq \Gamma$ be the set of AFDs in the path between f and f'. Then, the accuracy of ω is

$$acc(\omega) = \prod_{\gamma \in \Gamma'} acc(\gamma)$$

Example 15. *Consider q_1 from Example 7, which measures the average amount of weight lifted by elderly athletes per city and type of exercise. Let ω be a drill-down from* Facility.City*, which produces the following md-query:*

$q_2 = \langle\ \{\text{\_id}, \text{Exercises.Type}\}, (\text{User.Age} \geq 60), \text{Exercises.Sets.Weight}, \mathit{avg}\ \rangle$

Drilling down from Facility.City *to* _id *implies the navigation of* Facility.Name $\rightsquigarrow$ Facility.City *(with accuracy 0.98) and* _id $\rightarrow$ Facility.Name *(with accuracy is 1). Thus, it is $acc(\omega) = 0.98 * 1 = 0.98$.*

7. Experimental evaluation

Before presenting an experimental evaluation of our approach in terms of effectiveness and efficiency, we briefly discuss the organization of the metadata collected throughout the different phases. As shown in Figure 1, each phase of the approach produces and/or consumes some metadata: global and local schemas, mappings, and dependency graphs. An appropriate modeling of these

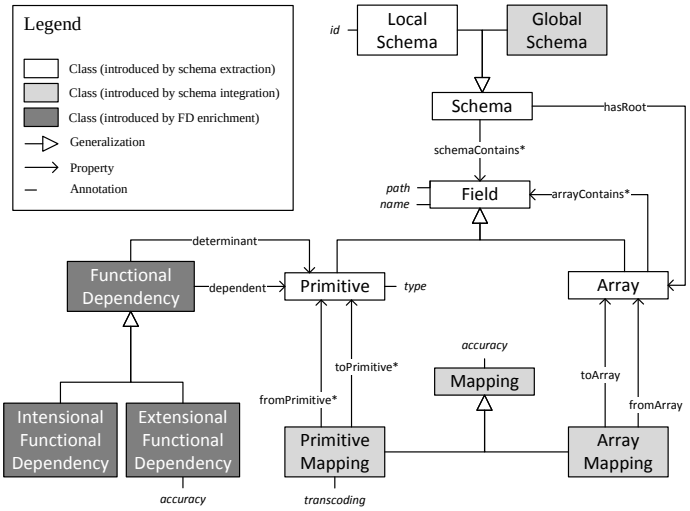

Figure 7: Metamodel of our approach

metadata is important to enable the interoperability of these phases. Since schemas are trees (Definition 2) while dependency graphs are DAGs (Definition 9), a natural choice is a database for graph data modeling. In particular, we rely on the triplestore of the Apache Jena Java framework to model metadata in RDF. Figure 7 shows the metamodel; boxes represent classes, while arrows represent different kinds of relationships between classes (i.e., standard generalizations, custom properties, and annotations). The use of the asterisk (*) in the property name is used to indicate a multiplicity higher than one. The color of the classes is representative of the phase in which they are first instantiated: white for schema extraction, light grey for schema integration, and dark grey for FD enrichment.

7.1. Efficiency

As a proof of concept for our approach we have developed some Java prototypes to support the main phases and tested them on two reference environments. The first one consists of a single Windows 7 machine, with a 4-core i7-2600 CPU @3.40 GHz and 16 GB of RAM; the second one is a cluster of seven CentOS 6 machines with an 8-core i7-4790 CPU @3.60 GHz and 32 GB

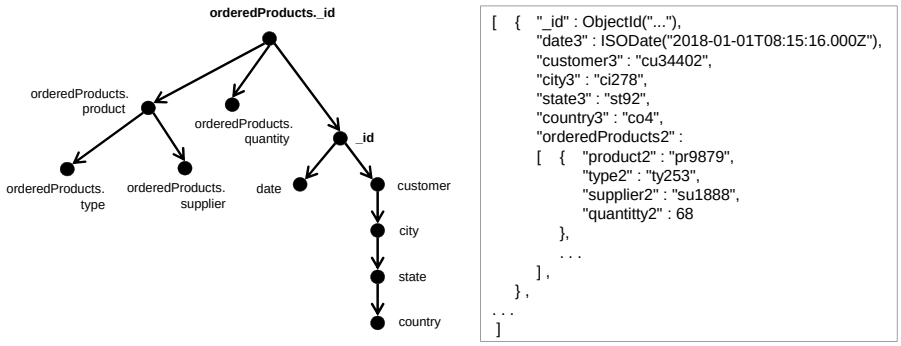

Figure 8: Excerpt of the dependency graph (left) for the synthetic collection Orders (right)

of RAM. Our reference real-world collection, WS, is stored on a Mongo DB 3.4 and randomly sharded on the cluster; it contains 5 M workout sessions with 6 different local schemas (mostly due to missing fields), 35 M exercises and 85 M sets. For a variety-aware evaluation, we also built two synthetic collections. The first one is called Orders and simulates a collection of orders of products by customers; it contains 5 M documents with 160 different local schemas (mostly due to variety in the field names and comprising a total of 150 different fields). Figure 8 shows a sample document and the derived dependency graph. The second synthetic collection is called HighlyNested and simulates a scenario in which each document has many arrays nested into each other; it contains 5 M documents with only 1 local schema, while the nested levels contain 15 M, 45 M, 135 M, and 405 M documents, respectively.

The prototypes are focused on schema extraction, AFD detection and query execution. Schema integration is currently done manually for WS (given the low number of schemas in our reference collection, this is still feasible) and automatically for Orders (which is feasible given the synthetic nature of the collection). Indeed, from the performance point of view, schema matching is not a particularly demanding task: as reported in [11], the schema matching tool COMA (available at `https://dbs.uni-leipzig.de/Research/coma.html`) can perform 468 millions root-based match comparisons (i.e., considering the full path of each field) in 41 seconds. This is quite promising, considering that the num-

Table 3: Execution times for schema extraction

Collection	# records	DB size	Time (standalone)	Time (cluster)
WS	5 K	2 MB	4 sec	3 sec
	50 K	20 MB	33 sec	19 sec
	500 K	197 MB	6 min	3 min
	5 M	1.7 GB	60 min	32 min
Orders	5 K	2 MB	1 sec	1 sec
	50 K	24 MB	10 sec	6 sec
	500 K	245 MB	2 min	1 min
	5 M	2.4 GB	19 min	10 min
HighlyNested	5 K	25 MB	11 sec	9 sec
	50 K	253 MB	2.5 min	1.5 min
	500 K	2.5 MB	28 min	15 min
	5 M	25 GB	4.7 hr	2.5 hr

ber of comparisons in our working example would not exceed the order of the tens of thousands. As to querying we remark that formulation, translation to MongoDB, evaluation, and evolution are all done in negligible time. Reformulation is done with polynomial complexity [2].

Schema extraction. Our implementation of schema extraction is loosely inspired by the free tool variety.js and consists of a simple routine that connects to the desired collection on MongoDB, extracts the local schemas, and writes the results on the triplestore. Execution times have been measured on the whole collection as well as on smaller samples to evaluate scalability. Times are shown in Table 3. In both reference environments, the time increases linearly with the size of the database, while the parallel architecture halves the times of the standalone environment. We also observe that times are consistent with those of related approaches that perform schema extraction on JSON datasets, such as [5].

A comparison between WS and the synthetic datasets shows that the time for schema extraction is independent of the number of local schema, but depends on the complexity of the documents; with respect to WS, the execution times on Orders decrease even if the number of local schemas is higher (because the documents are simpler) and increase on HighlyNested even if the number of local schemas is lower (because the documents are more complex).

Table 4: AFDs detected from one primitive (rows) to another (columns) in the WS dataset; cells set to TRUE are in green

f	User.FullName	User.LastName	Facility.Name	User.FirstName	Ex.ExCalories	Ex.Sets.SetCalories	Facility.City	Ex.Sets.Weight	DurationMins	StartedOn	Facility.Chain	Ex.Type	User.Age	Ex.Sets.Reps	Facility.Country	SessionType
User.FullName	-	×	×	×	(b)	(b)	×	(b)	×	×	×	(b)	✓	(b)	×	×
User.LastName	(a)	-	×	×	(b)	(b)	×	(b)	×	×	×	(b)	×	(b)	×	×
Facility.Name	(a)	(a)	-	×	(b)	(b)	✓	(b)	×	×	✓	(b)	×	(b)	(c)	×
User.FirstName	(a)	(a)	(a)	-	(b)	(b)	×	(b)	×	×	×	(b)	×	(b)	×	×
Ex.ExCalories	(a)	(a)	(a)	(a)	-	(b)	×	(b)	×	×	×	×	×	(b)	×	×
Ex.Sets.SetCalories	(a)	(a)	(a)	(a)	(a)	-	×	×	×	×	×	×	×	×	×	×
Facility.City	(a)	(a)	(a)	(a)	(a)	(a)	-	(b)	×	×	×	(b)	×	(b)	✓	×
Ex.Sets.Weight	(a)	(a)	(a)	(a)	(a)	(a)	(a)	-	×	×	×	×	×	×	×	×
DurationMins	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	-	×	×	(b)	×	(b)	×	×
StartedOn	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	-	×	(b)	×	(b)	×	×
Facility.Chain	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	-	(b)	×	(b)	×	×
Ex.Type	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	×	-	×	(b)	×	×
User.Age	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	-	(b)	×	×
Ex.Sets.Reps	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	-	×	×
Facility.Country	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	-	×
SessionType	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	-

AFD detection. This phase is done in two steps starting from the global schema in Figure 3. At first, the cardinality of each primitive field (excluding ids, for which intensional FDs are directly inferred) is determined via a call to *CountDistinct* in Algorithm 1, aimed at sorting the fields by descending values (ranging from over 400 thousand for User.FullName to 3 for SessionType). The times required mostly depend on the nesting level of the field: as shown in Table 5, fields in the root of the document took 1 to 5 seconds, fields in the Exercise array took about 32 seconds, while fields in the Exercise.Sets array took 50 to 55 seconds.

Then, Algorithm 1 is run with $\epsilon = 0.05$; Table 4 shows the AFD matrix Z, with cells set to true in green. Specifically, among true cells, for those marked with (c) the existence of the AFD has been inferred by the transitivity rule listed in Section 5 (the only case is Facility.Name $\rightsquigarrow$ Facility.Country), while for those marked with ✓ the check has been done and the AFD has been shown to hold (e.g., Facility.Name $\rightsquigarrow$ Facility.Chain holds with accuracy 0.98, as reported in Example 5). Among false cells, those marked with (a) and (b) indicate that the

Table 5: Query execution times (in seconds) for checking AFDs from one primitive (rows) to another (columns); the times for the queries avoided are in grey. The table also shows the cardinality of each primitive and the time (in seconds) for calculating it

f	$\|f\|$	*Time for distinct count*	User.FullName	User.LastName	Facility.Name	User.FirstName	Ex.ExCalories	Ex.Sets.SetCalories	Facility.City	Ex.Sets.Weight	DurationMins	StartedOn	Facility.Chain	Ex.Type	User.Age	Ex.Sets.Reps	Facility.Country	SessionType
User.FullN	416K	5	-	12	11	11	70	110	10	108	10	10	9	104	10	96	11	10
User.LastN	228K	4	12	-	5	5	53	89	4	79	4	3	3	66	2	81	2	2
Fac.Name	2134	1	11	5	-	3	42	67	3	66	2	2	2	39	2	63	2	2
User.FirstN	1845	3	11	5	3	-	40	68	3	67	2	2	2	39	2	62	2	2
Ex.ExCal	803	33	70	53	42	40	-	68	39	67	37	38	37	39	38	67	37	37
Ex.S.SetCal	515	54	110	89	67	68	68	-	68	75	67	67	66	66	65	70	65	64
Fac.City	298	1	10	4	3	3	39	68	-	70	2	2	2	37	1	62	1	1
Ex.S.Wght	204	54	108	79	66	67	67	75	70	-	66	65	65	65	65	68	63	63
DurMins	160	1	10	4	2	2	37	67	2	66	-	1	1	36	1	60	1	1
StartedOn	122	1	10	3	2	2	38	67	2	65	1	-	1	36	1	61	1	1
Fac.Chain	103	1	9	3	2	2	37	66	2	65	1	1	-	37	1	61	1	1
Ex.Type	100	32	104	66	39	39	39	66	37	65	36	36	37	-	40	62	39	39
User.Age	70	1	10	2	2	2	38	65	1	65	1	1	1	40	-	60	1	1
Ex.S.Reps	40	53	96	81	63	62	67	70	62	68	60	61	61	62	60	-	74	73
Fac.Country	5	1	11	2	2	2	37	65	1	63	1	1	1	39	1	74	-	1
SessType	3	1	10	2	2	2	37	64	1	63	1	1	1	39	1	73	1	-

existence of the AFD has been excluded by the first and second pruning rule in Section 5, respectively, while for those marked with $\times$ the AFD has been checked and has turned out not to hold. The only case when a check has to be done for an inverse AFD is for Exercise.Type $\rightsquigarrow$ Facility.Chain, because Facility.Chain $\rightsquigarrow$ Exercise.Type was excluded thanks to rule (b) and $1 - \frac{|\text{Exercise.Type}|}{|\text{Facility.Chain}|} = 1 - \frac{100}{103} = 0.03 \leq \epsilon$. Overall, the rules of Algorithm 1 allow to save 35 AFD checks, i.e., 29% of the 120 checks above the diagonal.

The times (in seconds) required to run each call to *CountDistinct* is shown in Table 5. Note that this table reports the times for every couple of primitives and in both directions; this is done to simplify the read of the table and to quantify the time saving due to the pruning rules that exclude checking some AFDs. In particular, consider that if $f \rightsquigarrow f'$ is checked (i.e., $CountDistinct(f_i, f_j)$ is called) then checking $f' \rightsquigarrow f$ is immediate (i.e., it does not require to call $CountDistinct(f_j, f_i)$, because its result is clearly the same as $CountDistinct(f_i, f_j)$).

Table 6: AFDs detected from one primitive (rows) to another (columns) in the Orders dataset; cells set to TRUE are in green

f	customer	op.product	op.supplier	city	date	op.type	state	country	op.category	continent
customer	-	(b)	(b)	✓	×	(b)	(c)	(c)	(b)	(c)
op.product	(a)	-	✓	×	×	✓	×	×	(c)	×
op.supplier	(a)	(a)	-	×	×	×	×	×	×	×
city	(a)	(a)	(a)	-	×	(b)	✓	(c)	(b)	(c)
date	(a)	(a)	(a)	(a)	-	(b)	×	×	(b)	×
op.type	(a)	(a)	(a)	(a)	(a)	-	×	×	✓	×
state	(a)	(a)	(a)	(a)	(a)	(a)	-	✓	(b)	(c)
country	(a)	(a)	(a)	(a)	(a)	(a)	(a)	-	(b)	✓
op.category	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	-	×
continent	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	(a)	-

First of all, Table 5 shows that the execution times are influenced by two factors: the nesting level of the involved primitives and the respective cardinalities. The time increases significantly when fields with high cardinality are involved; indeed, the highest execution times (almost 2 minutes) are registered on AFD checks from User.FullName (i.e., the primitive with the highest cardinality) to primitives in the Exercise.Sets array (the array with the highest nesting level).

Remarkably, thanks to the pruning rules of Algorithm 1, the call to $CountDistinct$ is actually skipped for many couples of fields in both directions. Interestingly, the skipped calls are often among the most expensive ones. Indeed, the overall cost for checking the AFDs drops from 69 minutes (if no rule were applied) to 32 minutes, with a 54% reduction in execution time.

To evaluate the impact of the transitivity rule (c) in Algorithm 1, Table 6 shows the AFD matrix Z for the Orders dataset (execution times range between 30 and 95 seconds). Since the dataset is synthetic, we simulate the best case scenario, in which all detected FDs are exact, so that the transitivity rule can always be applied. In this case, the rule is applied 7 times for a total time saving of 305 seconds, i.e., a 15% reduction in execution time. In a real scenario, this value may be more or less attenuated by the presence of AFDs that prevent the rule to be applied. Nonetheless, this demonstrates that its impact on the performance can be significant.

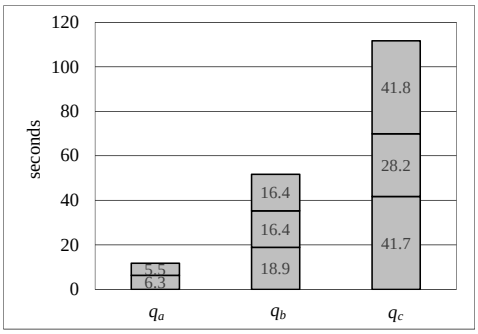

Figure 9: Execution times of three queries (q_a, q_b and q_c) split into the execution times of the required local queries

Query execution. Consider the following three md-queries:

$q_a = \langle\ \{\mathsf{User.Age}, \mathsf{Facility.Chain}\}, TRUE, \mathsf{DurationMins}, \mathtt{avg}\ \rangle$

$q_b = \langle\ \{\mathsf{User.FullName}\}, (\mathsf{SessionType} = \text{“AdvancedProgram”}), \mathsf{Exercises.ExCalories}, \mathtt{sum}\ \rangle$

$q_c = \langle\ \{\mathsf{Facility.Name}, \mathsf{Exercises.Type}\}, (\mathsf{StartedOn} \geq 01/01/2018), \mathsf{User.Age}, \mathtt{max}\ \rangle$

Due to the reformulation on the local schemas, 6 local md-queries are created from each of the three global ones. A simple optimization is done, when possible, to merge the local queries that involve the same fields on different local schemas; for instance, with reference to Figure 3, a query counting the documents by SessionType can be translated into a single local query, as every local schema has the same representation of SessionType. Due to this optimization, q_a, q_b, and q_c are reformulated into either 2 or 3 local queries. The execution times in seconds for each query (subdivided into the times for each local query) are shown in 9; the times for q_b and q_c are higher due to the necessity of unwinding arrays.

7.2. Effectiveness

In this section we focus on the querying phase to evaluate the effectiveness of our approach. In the WS collection, the 5 M workout sessions are distributed among the six schemas as indicated in Table 7; the schemas of s_1 and s_2 are the ones shown in Figure 3, while the others are variations of these two.

Table 7: Cardinality of local schemas for the WS collection

Local schema s_i	$\|D^{s_i}\|$	Percentage
s_1	2208852	44%
s_2	1446653	29%
s_3	825592	17%
s_4	349874	7%
s_5	113356	2%
s_6	39319	1%

The goal of these tests is to measure how querying benefits from our approach compared to a *plain* scenario, in which no schema integration has been done. Consider q_a, q_b and q_c as defined in the closing part of Section 7.1. With no integration and reformulation, these queries could only hit the fields that share the exact pathname. In this case we can reasonably assume that they would be formulated on the fields that exist in s_1, i.e., the local schema with the highest cardinality. We simulate a progressive integration of the other local schemas (in the same order as they are listed in Table 7) and evaluate the variation of different metrics: the global support of the fields involved in the query and the indicators of query level and measure densities. These variations are shown in Figure 10, where each chart corresponds to a single query.

First, we observe that the benefits vary from query to query. For instance, $levelDensity(q_a)$ and $measureDensity(q_c)$ are not affected by the progressive integration, because they involve fields (i.e., User.Age and Facility.Chain) that are always used with the same name and path. Remarkably, in the two other cases there is a significant increase of the global support of most fields and, consequently, of the query indicators.

7.3. Case study

We conclude our experimental evaluation by presenting a case study that simulates an OLAP session by the user, in order to illustrate the benefits of the indicators. Let the user begin the session with the following md-query, which retrieves the average user age in the different facility chains:

$$q' = \langle \{\mathsf{Facility.Chain}\}, TRUE, \mathsf{User.Age}, \mathsf{avg} \rangle$$

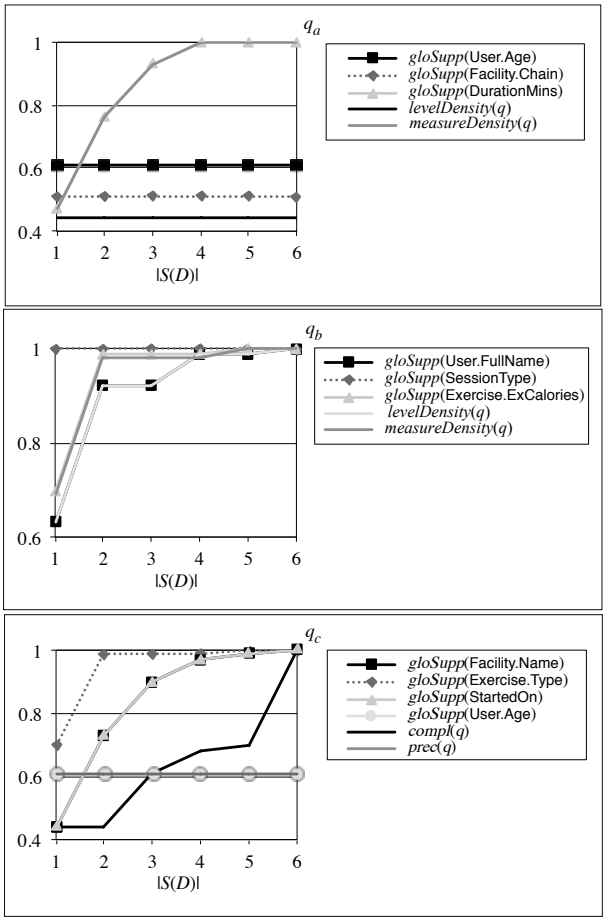

Figure 10: Effectiveness for three md-queries, measured in terms of global support, level density, and measure density

Along with the results, the indicators show that $levelDensity(q') = 0.51$ and $measureDensity(q') = 0.61$. The low values of both densities tell the users that either some mappings are missing (in that case, the user can go back to the schema integration stage to check the local schemas and possibly add some mapping she had overlooked) or the data is incomplete. In both cases, the results are not really useful, as many values are missing and those that are present may have fallen into uncertain groups (i.e., those for which a balancing strategy is applied).

To improve the quality of the results, the user formulates a new query

q'' trying to filter out the session for which the data is missing. One option is to exclude short-lasting sessions (i.e., where DurationMins $\geq$ 30); the other is to consider only sessions of a given type (e.g., where SessionType = "AdvancedProgram"). Given the presence of a profiling rule on SessionType, the user knows in advance that the second filter will lead to $levelDensity(q'') = 1$ and $measureDensity(q'') = 0.73$, because the query will be restricted to local schemas s_1 (where both Facility.Chain and User.Age are present) and s_4 (where only Facility.Chain is present). Thus, the user issues the following md-query:

$$q'' = \langle \{\text{Facility.Chain}\}, \text{SessionType} = \text{"AdvancedProgram"}, \text{User.Age}, \text{avg} \rangle$$

Although the measure density is not substantial, the level density is high enough to guarantee a reliable reading of the results, as no values have been aggregated in uncertain groups. Moreover, group measure density helps the user in identifying the groups for which the result is most reliable: among the 103 chains, over 30% of them have a group measure density greater than 0.95 in q''. An analytical tool may display the results with a heat map for an intuitive reading.

Finally, the user wants to see the results from a different perspective by formulating a query q''' that drills down from Facility.Chain to Facility.Name:

$$q''' = \langle \{\text{Facility.Name}\}, \text{SessionType} = \text{"AdvancedProgram"}, \text{User.Age}, \text{avg} \rangle$$

Since $acc(\text{Facility.Name} \rightsquigarrow \text{Facility.Chain}) = 0.98$, the user is aware that the results of q''' are not completely consistent with those of q''. However, the high accuracy indicates that there is only a small error in the data, hinting to the fact that this could be simply due to the presence of facilities with the same name that belong to different chains. Ultimately, this points the user towards the correct decomposition, which can be obtained by grouping by both Facility.Chain and Facility.Name in query q'''':

$$q'''' = \langle \{\text{Facility.Chain}, \text{Facility.Name}\}, \text{SessionType} = \text{"AdvancedProgram"}, \text{User.Age}, \text{avg} \rangle$$

8. Related literature

The rise of NoSQL stores has captured a lot of interest from the research community, which has proposed a variety of approaches to deal with the schema-less feature. Early works had already focused on schema discovery from web data [21] and XML objects [22], but the attention has now shifted to the widely adopted JSON format and to key/value repositories in general. Dealing with schemaless sources often requires the adoption of data integration techniques [23] to provide a unified view of the diverse available data —much like in the integration or federated querying of different databases [2, 24]. As this is not the primary focus of the paper, we refer the reader to a recent survey on the subject [25].

A first distinction from the closely related works lies in how each of them approaches the problem of schema discovery. Some works aim at providing a comprehensive view of the schema variety in JSON documents; e.g., [7] proposes a reverse engineering process to derive a versioned schema model, where multiple versions of the same field are created for every intensional variation detected in the collection. Other works provide a more concise representation that tends to hide schema variety. For instance, [6] couples a clustering technique with schema matching techniques to identify a *skeleton* containing the smallest set of core fields, [10] simply models the union of all the fields in a collection, while [5] adopts regular expressions to model the variability of a field type. Our work is closer to the latter group, although our global schema captures the entire variety of fields and enables the user to choose the fields to focus on, while assisting her with quality indicators of the final queries. Several free tools have also been released to perform schema detection on different platforms (MongoDB, ElasticSearch, Couchbase, Apache Drill), although they are mostly limited to collecting the union of the fields. In a previous work [1] we followed a different approach and devised a *schema profiling* algorithm that explains the schema variety in a collection in terms of the extensional values found in the documents (e.g., it could find that different schemas depend on the different values for

SessionType).

The most distinguishing feature of our approach is the definition of a multidimensional representation of the schema in order to enable OLAP analyses directly on the DOD. From this point of view, a work closely related to ours is [15], which proposes a schema-on-read approach for md-queries over DODs. They build a multidimensional schema from the union of fields found in the collection; then, the OLAP experience is proposed at query time, giving suggestions for roll-up and drill-down operations based on the last query formulated by the user. The work in [15] differs from our under several aspects:

- [15] exclusively focuses on the multidimensional representation of JSON data and overlooks the schemaless property of DODs: in particular, inter-document variety is considered only in terms of fields with varying support. Conversely, our approach proposes the schema extraction and schema integration phases to fully capture inter-document variety and maximize the support of each field.

- AFD detection is covered by both approaches; however, in [15] it is activated on demand only *after* the user has written a query. We believe this represents a limitation, as the user discovers hierarchies only by issuing queries. Also, [15] proposes pruning rules for AFD detection, but our strategy improves them by also taking transitivity into consideration.

- As to querying, [15] provides limited support to the OLAP experience (due to the aforementioned absence of an integration step and of a comprehensive FD enrichment), while no support is given to the user to evaluate queries and OLAP operations in terms of level density, measure density and accuracy.

Another similar work is [26], which proposes a MapReduce-based algorithm to compute OLAP cubes on columnar stores. The approach is meant to work on a data warehouse (i.e., a database already comprising facts and dimensions); besides, it is limited to the computation of the cubes, while the querying aspect

is mentioned as future work. Also [27] aims at delivering the OLAP experience, but its operational data source is a graph-based database, whose data model is entirely different from the one of DODs. Finally, [28] builds on [6] to propose a complete architecture that ingests NoSQL data and provides schema-on-read functionalities, but without mentioning multidimensional enrichment and OLAP analyses.

On the issue of schema variety on DODs, [29] is a recent work that builds on a simple mapping strategy to hide the variety within a single, comprehensive query. Whereas the approach promisingly proposes a simpler querying mechanisms, it only covers a limited set of schema variants and does not support the OLAP scenario.

Since schema variety in a collection often consists of different representation of the same data (e.g., due to schema evolution or to the ingestion of data from different sources), the problem of schema discovery is often coupled with schema matching algorithms. [30] provides a comprehensive summary of the different techniques envisioned for generic schema matching (which ranges from the relational world to ontologies and XML documents); it is mentioned as a baseline reference in [6], while [31] starts from there to define its own algorithm for schema matching on NoSQL stores based on subtree matching. In [32] a tool is presented to automatically identify evolution in the schema of instances in NoSQL databases: once a schema change is detected, the tool either updates the database instances to enforce schema consistency or provides a code to deal with this issue on the application side. This structured approach differs from our schema-on-read scenario, which transparently handles schema differences and avoids to update the original data.

Several works have focused on bringing NoSQL back to the relational world. [33] discusses an approach to provide schema-on-read capabilities for *flexible schema* data stored on RDBMSs; this is done by mapping the document structure on different tables and by providing a *data guide* as the union of every possible field at any level. Differently from our approach, no advanced schema matching mechanism is provided. [34] proposes an algorithm to provide a generic

relational encoding of arbitrary JSON documents; in particular, documents are stored in ternary relations that contain rows for every key in every document (i.e., each row stores the document id, the key name, and the key value). A more sophisticated algorithm is proposed in [31], where normalized relational schemas are automatically generated from NoSQL stores. It relies on AFD detection to build relationships between entities and it provides its own schema matching algorithm. Based on this approach, a vision for a new paradigm called *adaptive schema databases* has been proposed in [35]; it is a conceptual framework that devises global schemas as time-evolving and user-dependent relational views that are mapped to local schemas via probabilistic mappings —whereas mappings are deterministic in our approach.

The work we present in this paper is based on [3], where the approach was originally motivated and introduced. Here we have improved our previous work under several substantial aspects: (i) the integration method we adopt to build the global schema is explained; (ii) the procedure to explore the space of all possible AFDs is formalized; (iii) the algorithms for the composition of the MongoDB queries are formalized; (iv) the discussion of the query indicators has been extended to evaluate both the whole query and the single groups in the result and to predict the value of the indicators before executing the query (to improve the accuracy of the prediction, also the possibility of adopting a schema profiling technique [1] is considered); (v) an experimental evaluation is presented to discuss the prototypical environment and to evaluate the schema extraction and FD enrichment phases in terms of efficiency and effectiveness.

9. Conclusions

In this paper we have presented an original approach to OLAP on DODs. Our basic claim is that the heterogeneity and schema variety intrinsic to DODs should be considered as a source of information wealth and showed to users together with quality indicators that assess its impact. At the core of our approach are (i) the building of a global schema that maps onto the different

local schemas within a collection, (ii) the translation of this schema into multidimensional form enhanced by the detection of approximate FDs, and (iii) the reformulation of queries from the global schema onto the local schema to improve the completeness of the result. After describing in detail the different phases of our approach, we have experimentally evaluated it from the points of view of efficiency and effectiveness.

As part of our future work we plan to make several improvements to the approach, on both its theoretical grounds and its technological implementation. The first step will be to increase the efficiency of the querying phase. Though the query execution times depend on the performance of MongoDB, we argue that a more sophisticated optimization of query reformulation may reduce the impact of issuing separate queries for each local schema. In this direction, we will evaluate techniques such as [29] that overcome the same problem by issuing a single but more complex query. Another aspect to consider is the introduction of online repairing of AFDs [36, 37, 38]: the idea is to automatically repair the errors in AFDs at query time, so that the user can directly receive the correct results without the need to modify the original data. Another direction is to broaden the scope of the approach to a scenario with multiple collections, thus extending the support to the whole DOD. On the implementation side, we aim to build a fully working prototype. Also, since DODs usually provide connectors to big data tools (e.g., Apache Spark), we will evaluate alternative query languages and execution engines to enhance the performance and expressiveness of the approach.

References

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

[2] M. Golfarelli, F. Mandreoli, W. Penzo, S. Rizzi, E. Turricchia, OLAP query reformulation in peer-to-peer data warehousing, Inf. Syst. 37 (5) (2012) 393–411.

[3] E. Gallinucci, M. Golfarelli, S. Rizzi, Variety-aware OLAP of document-oriented databases, in: Proc. DOLAP, Vienna, Austria, 2018.

[4] O. Romero, A. Abelló, Multidimensional design by examples, in: Proc. DaWaK, Krakow, Poland, 2006, pp. 85–94.

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

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

[7] 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.

[8] J. L. C. Izquierdo, J. Cabot, Discovering implicit schemas in JSON data, in: Proc. ICWE, 2013, pp. 68–83.

[9] C. Batini, M. Lenzerini, S. Navathe, A comparative analysis of methodologies for database schema integration, ACM Computing Surveys 18 (4) (1986) 323–364.

[10] M. Klettke, U. Störl, S. Scherzinger, O. Regensburg, Schema extraction and structural outlier detection for JSON-based NoSQL data stores., in: Proc. BTW, Vol. 2105, 2015, pp. 425–444.

[11] S. Maßmann, S. Raunich, D. Aumüller, P. Arnold, E. Rahm, Evolution of the COMA match system, in: Proceedings of the 6th International Workshop on Ontology Matching, Bonn, Germany, October 24, 2011, 2011.

[12] 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.

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

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

[15] M. L. Chouder, S. Rizzi, R. Chalal, EXODuS: Exploratory OLAP over document stores, Inf. Syst. 79 (2019) 44–57.

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

[17] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, What can hierarchies do for data warehouses?, in: Proc. VLDB, Edinburgh, Scotland, 1999, pp. 530–541.

[18] M. Golfarelli, S. Rizzi, Data warehouse design: Modern principles and methodologies, McGraw-Hill, Inc., 2009.

[19] H. Lenz, A. Shoshani, Summarizability in OLAP and statistical data bases, in: Proc. SSDBM, 1997.

[20] F. Naumann, J. C. Freytag, U. Leser, Completeness of integrated information sources, Inf. Syst. 29 (7) (2004) 583–615.

[21] S. Nestorov, J. Ullman, J. Wiener, S. Chawathe, Representative objects: Concise representations of semistructured, hierarchical data, in: Proc. ICDE, 1997, pp. 79–90.

[22] J. Hegewald, F. Naumann, M. Weis, XStruct: Efficient schema extraction from multiple and large XML documents, in: Proc. ICDE Workshops, 2006, pp. 81–81.

[23] M. Lenzerini, Data integration: A theoretical perspective, in: Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on

Principles of Database Systems, June 3-5, Madison, Wisconsin, USA, 2002, pp. 233–246.

[24] J. Koh, A. L. P. Chen, Efficient query processing in integrated multiple object databases with maybe result certification, IEEE Trans. Knowl. Data Eng. 14 (4) (2002) 691–708.

[25] B. Golshan, A. Y. Halevy, G. A. Mihaila, W. Tan, Data integration: After the teenage years, in: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017, 2017, pp. 101–106.

[26] K. Dehdouh, Building OLAP cubes from columnar NoSQL data warehouses, in: Proc. MEDI, Almería, Spain, 2016.

[27] A. Castelltort, A. Laurent, NoSQL graph-based OLAP analysis, in: Proc. KDIR, Rome, Italy, 2014, pp. 217–224.

[28] R. Hai, S. Geisler, C. Quix, Constance: An intelligent data lake system, in: Proc. SIGMOD, San Francisco, USA, 2016, pp. 2097–2100.

[29] H. B. Hamadou, F. Ghozzi, A. Péninou, O. Teste, Towards schema-independent querying on document data stores, in: Proceedings of the 20th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data co-located with 10th EDBT/ICDT Joint Conference (EDBT/ICDT 2018), Vienna, Austria, March 26-29, 2018., 2018.

[30] E. Rahm, P. A. Bernstein, A survey of approaches to automatic schema matching, VLDB J. 10 (4).

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

[32] S. Scherzinger, E. C. de Almeida, T. Cerqueus, L. B. de Almeida, P. Holanda, Finding and fixing type mismatches in the evolution of object-NoSQL mappings, in: Proc. Workshops EDBT/ICDT, 2016.

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

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

[35] W. Spoth, B. S. Arab, E. S. Chan, D. Gawlick, A. Ghoneimy, B. Glavic, B. C. Hammerschmidt, O. Kennedy, S. Lee, Z. H. Liu, X. Niu, Y. Yang, Adaptive schema databases, in: Proc. CIDR, Chaminade, USA, 2017.

[36] Y. Gao, X. Miao, Query Processing over Incomplete Databases, Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2018.

[37] S. De, Y. Hu, Y. Chen, S. Kambhampati, Bayeswipe: A multimodal system for data cleaning and consistent query answering on structured bigdata, in: 2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014, 2014, pp. 15–24.

[38] J. García-García, C. Ordonez, Repairing OLAP queries in databases with referential integrity errors, in: DOLAP 2010, ACM 13th International Workshop on Data Warehousing and OLAP, Toronto, Ontario, Canada, October 30, 2010, Proceedings, 2010, pp. 61–66.