Variety-Aware OLAP of Document-Oriented Databases

Enrico Gallinucci
DISI—Univ. of Bologna, Italy
Cesena, Italy
enrico.gallinucci2@unibo.it

Matteo Golfarelli
DISI—Univ. of Bologna, Italy
Cesena, Italy
matteo.golfarelli@unibo.it

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

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 related for instance to query completeness and precision.

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 attributes, in different names or types for an attribute, and in different structures for instances [9]. 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. 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, e.g., in terms of completeness and precision. 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 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 attributes, different levels of detail in instances, different attribute naming.
- Our approach to reformulation of multidimensional queries on heterogeneous documents grounds on a formal approach [11], which ensures its correctness and completeness.
- We propose a set of indicators to make the user aware of the level of completeness and precision of the query result.

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 attributes are either included or absent, but rather to enable OLAP querying on some sort of "soft" schema where each source attribute 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 discuss the related literature, and finally in Section 8 we draw the conclusions. An appendix completes the paper discussing the correctness of our query reformulation framework.

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

© 2018 Copyright held by the owner/author(s). Published in the Workshop Proceedings of the EDBT/ICDT 2018 Joint Conference (March 26, 2018, Vienna, Austria) on CEUR-WS.org (ISSN 1613-0073). Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

CEUR-WS.org/Vol-2062/paper02.pdf

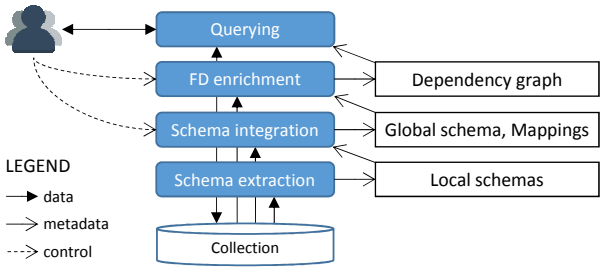

Figure 1: Approach overview

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 approach, that allows users to manually validate/refine the mappings proposed by the system. As of now, we rely on the user to manually provide inter-schema mappings, from which the global schema is derived.

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 attributes. 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 [19]. 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 in terms of quality and reliability of the results, we show her a set of indicators related to query completeness and precision.

The motivation example that we use across the paper is based on a real-world collection of workout sessions,

```
[ { "_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

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, 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 3.1 (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 3.2. 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 [1, 24] of the most frequent fields, or by keeping track of every different schema [20]. 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 [1, 15], 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 3.3 (Schema). A *schema* is a directed tree $s = (F, A)$ 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 the root r of s), 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 $(a, f) \in A$.

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 3.4 (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 and with type array.

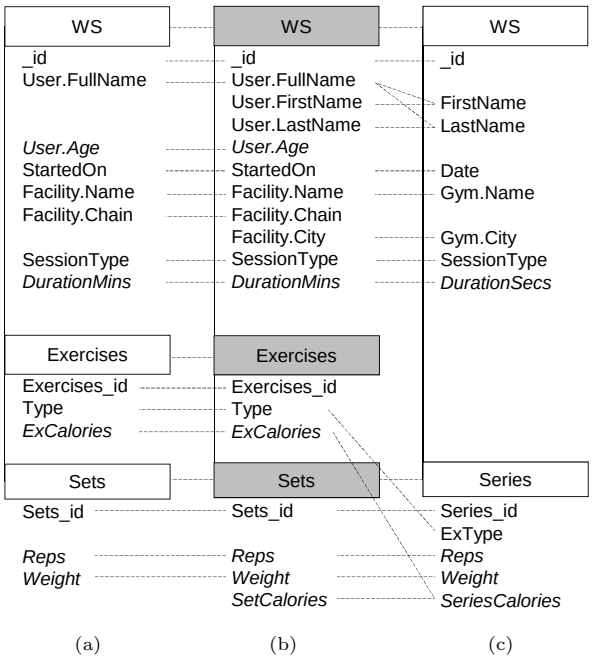

Figure 3: The schema of the JSON document in Figure 2 (a), another schema of the same collection (c), and the global schema (b)

(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.

Example 3.5. Figure 3.a shows the schema 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 $arr(\mathsf{Exercises.Type}) = \mathsf{Exercises}$ and $id(\mathsf{Exercises}) = \mathsf{Exercises.\_id}$. □

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 and the same type).

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

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. The global schema

can be incrementally built using one of the methodologies discussed in [2]; for instance, adopting a *ladder* integration strategy, 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, notice that some mappings may be missed by adopting a purely incremental strategy (i.e., a second iteration on the local schemas may be required). A survey of the techniques that can be used for finding mappings is provided in [3, 18].

A mapping is defined as follows:

Definition 4.1 (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*, $\phi : Dom(P) \rightarrow Dom(P')$.

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

Definition 4.2 (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 $\langle r_i, r \rangle$ between the roots of s_i and $g(D)$;
(2) every field f in each s_i is involved in at least one mapping onto the fields of $g(D)$;
(3) every field f in $g(D)$ is involved in at least one mapping with some s_i.

Example 4.3. Figure 3 shows two sample schemas from the WS collection (a and c) and the corresponding global schema $g(D)$ (b); mappings are represented with dotted lines. An example of array mapping from local schema (c) to global schema (b) 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 Appendix). 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 the second schema 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 = Exercises.ExCalories in the global schema, $arr(f)$ is Exercises in the schema of Figure 3.a and Exercises.Sets in the schema of Figure 3.c), 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 4.4 (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:

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

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 4.5 (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 4.6. In our working example, let the 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.City 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 $gloSupp(f) = 0.8 * 0.5 + 0.4 * 0.5 = 0.6$. □

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 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., _id → SessionType);
- 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., Exercises._id → _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 checking the data. More precisely, since DODs may contain incomplete and faulty data, we have to look for *approximate FDs* (AFDs), i.e., FDs that "mostly" hold on data —like done for instance in [6, 10, 23].

Definition 5.1 (Approximate Functional Dependency). Given two fields f and f', let $acc(f, f') \in [0..1]$ denote the ratio between the number of unique values of f and the number of unique values of (f, f'). We will say that AFD $f \rightsquigarrow f'$ holds if $acc(f, f') \geq \epsilon$, where ϵ is a user-defined threshold [14].

To detect AFDs and create hierarchies accordingly, some approaches that were recently devised in the literature (e.g., [10] and [6]) can be reused, possibly coupled with traditional approaches to multidimensional modeling based on FDs (e.g., [23]). Interestingly, in [6] the number of checks to be made for AFD detection is effectively reduced thanks to the intensional FDs provided by the global schema. Note that, differently from [6], in our approach we consider inter-document variety, so the queries that check for AFDs must be reformulated from the global schema on each local schema. How this can be done is discussed in the Appendix.

Definition 5.2 (Dependency Graph). Given the global schema $g(D) = (F, A)$ and an (acyclic) set of (A)FDs Γ, the *dependency graph* is a couple $\mathcal{M} = (F^{prim}, \succeq)$ where F^{prim} is the set of primitive nodes in F and $\succeq$ is a *roll-up* partial order of F^{prim} derived from Γ. In particular, $f_j \succeq f_k$ (i.e., f_j is a predecessor of f_k in $\succeq$) if either $f_j \rightsquigarrow f_k \in \Gamma$ or $f_j \rightarrow f_k \in \Gamma$.

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 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 5.3. Figure 4 shows the dependency graph for our working example. Each primitive field is represented as a circle whose color is representative of 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 gray). Note that, in this case, the dependency graph is a tree, because in the global schema of Figure 3.b arrays are nested within each other. A different situation is the one shown in Figure 5, where the collection includes documents with two arrays at the same level, so the dependency graph is not a tree. □

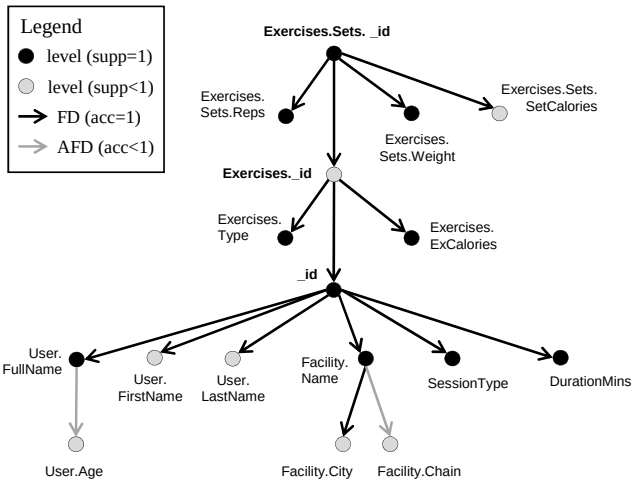

Figure 4: Dependency graph for the global schema in Figure 3.b

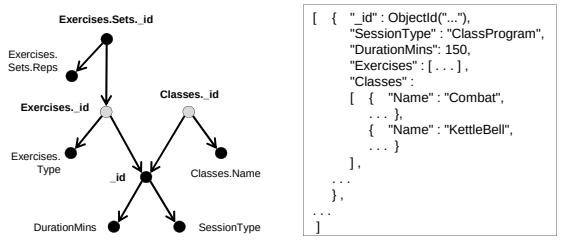

Figure 5: Excerpt of the dependency graph (left) in presence of alternative documents (right)

6 QUERYING

In this section we describe the final querying stage. We start by providing the definition of an OLAP query and discussing its correctness from a multidimensional standpoint (Section 6.1). Then, we discuss the execution of a query, which mainly involves its translation into the MongoDB language and the reformulation from the global schema to the local schemas (Section 6.2). Finally, we introduce a set of indicators to evaluate a query in the context of an OLAP session (Section 6.3).

6.1 Query Formulation

First of all, we define an OLAP query as follows.

Definition 6.1 (OLAP query). Given dependency graph $\mathcal{M} = (F^{prim}, \succeq)$, an OLAP 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 $f_j \not\succeq f_k$;
- 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`);

Algorithm 1 Validity check of an OLAP query

Input a dependency graph $\mathcal{M} = (F^{prim}, \succeq)$, an OLAP query $q = \langle G, p, m, \varphi \rangle$
Output a validity status
1: $warn \leftarrow$ false
2: **for each** $f \in G$ **do**
3: **if** $id(arr(m)) \not\succeq id(arr(f))$ **then**
4: $warn \leftarrow$ true ▷ Disjointness failed
5: **if** $gloSupp(f) < 1$ **then**
6: $warn \leftarrow$ true ▷ Completeness failed
7: **if** $warn$ **then**
8: **return** "warning"
9: **else**
10: **return** "valid"

- there exists in $\mathcal{M}$ one single field $\overline{f}$ such that $\overline{f} \succeq f$ for all other fields 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 5, with $G = \{\mathsf{Classes.Name}, \mathsf{Exercises.Type}\}$.

Example 6.2. The following query, 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 $fact(q_1) = \mathsf{Exercises.Sets.\_id}$. □

In [19] the authors outline the constraints that must hold for an OLAP 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* [16], 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 6.1 already ensures that 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 1) 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 [19]). 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 these information can hardly be inferred from the schema or even provided by the user [6].

Example 6.3. Query q_1 passes the validity check of Algorithm 1 with a completeness warning, because $gloSupp(\mathsf{Facility.City}) < 1$. On the other hand, q_1 meets the disjointness constraint because

$$id(arr(\mathsf{Facility.City})) = \mathsf{\_id}$$
$$id(arr(\mathsf{Exercises.Type})) = \mathsf{Exercises.\_id}$$
$$id(arr(\mathsf{Exercises.Sets.Weight})) = \mathsf{Exercises.Sets.\_id}$$
$$\mathsf{Exercises.Sets.\_id} \succeq \mathsf{\_id}$$
$$\mathsf{Exercises.Sets.\_id} \succeq \mathsf{Exercises.\_id}$$

□

As previously mentioned, a 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: *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 $l_k \succeq l_j$), and *upward balancing* (replacing with values from the closest level l_k such that $l_j \succeq l_k$) [12]. Whereas they are 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 a 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 Execution

Once a 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. How this can be done is discussed in the Appendix. In the remainder to this subsection we explain how, after reformulation, each single query obtained can be translated to MongoDB.

OLAP queries are translated to MongoDB according to its 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 stages are: `$match` (to apply predicate selections), `$project` (to apply transformations to the single fields), `$unwind` (to unfold an array by creating a different document for every object inside the array), `$group` (to group the documents and calculate aggregated values).

Given 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) For every array a in $g(D)$, $a \neq r$, for which there is a field f mentioned in q such that $fact(q) \succeq id(a) \succeq f$, an `$unwind` stage is defined; the order of this stages reflects the order of the arrays in $g(D)$, beginning from the one closest to r.
(2) If $p \neq \emptyset$, 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, in accordance with the balancing strategy chosen by the user. Additionally, 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.

(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 6.3.

Example 6.4. 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]
     } }
   } }
}
```

□

6.3 Query Evaluation and Evolution

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 an OLAP query after it has been executed. Let E be the set of distinct groups returned by query q; each group $e \in E$ includes $|e|$ objects (measured by the count field as of Section 6.2), of which $|e|_m$ (measured by the count-m field) have a value for m.

Selectivity. This indicator measures the selectivity of the selection predicates in q:

$$sel(q) = \frac{\sum_{e \in E} |e|}{|fact(q)|}$$

Completeness. This indicator is built on the concept of completeness previously introduced. The idea is to show the percentage of the queried objects that have not been affected by the balancing strategies (which steps in when the value of a level is null or does not exists):

$$compl(q) = \frac{\sum_{e \in E, !balanced(e)} |e|}{\sum_{e \in E} |e|}$$

where $balanced(e)$ is true if e has been balanced, false otherwise (as stated by the balanced field introduced in Section 6.2).

Group precision. While the absence of full support on levels can be overcome by the balancing strategies, nothing can be done when it involves the query measure. In this case, the precision of the aggregated value returned for each group is determined by the percentage of aggregated objects that actually contain a value for the measure. Thus, the precision of a group e is

$$prec(e) = \frac{|e|_m}{|e|}$$

Consistently with an OLAP scenario, a 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 $f \succeq f'$ (or $f' \succeq f$).

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:

Accuracy. 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. Given query q, let q' be the query resulting from a roll-up (or drill-down) of q 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 q' with respect to q is

$$acc(q', q) = 1 - \prod_{\gamma \in \Gamma'} acc(\gamma)$$

7 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 schemaless feature. In particular, most of the recent works focus on the widely adopted JSON format and on key/value repositories in general.

A first distinction lies in how each work approaches the problem of schema discovery. Some works aim at providing a comprehensive view of the schema variety in JSON documents; e.g., [20] 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, [24] couples a clustering technique with schema matching techniques to identify a *skeleton* containing the smallest set of core fields, while [1] 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 [9] 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 [6], which proposes a schema-on-read approach for OLAP queries over DODs. This is done by building a multidimensional schema from the union of fields found in the collection; then, the OLAP experience is proposed at query time, where suggestions for roll-up and drill-down operations are provided given the query formulated by the user. Differently from our approach, [6] 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 (thus no schema integration is performed), and no support is given to the user to evaluate the coverage and accuracy of queries. Also, AFD detection is carried out on demand only *after* the user has written a query, thus it only impacts the OLAP experience. Another similar work is [7], 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 OLAP querying aspect is mentioned as future work. Also [4] 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, [13] builds on [24] to propose a complete architecture that ingests NoSQL data and provides schema-on-read functionalities, but without mentioning multidimensional enrichment and OLAP analyses.

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. [18] 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 [24], while [8] starts from there to define its own algorithm for schema matching on NoSQL stores based on subtree matching. In [21] 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. [17] 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. [5] 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 [8], 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 [22]; 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.

8 CONCLUSIONS

In this paper we have presented an original approach to OLAP on DODs. Our current implementation relies on a prototype that separately handles the different stages. In

Table 1: Execution times for schema extraction

# records	DB size	Time
5 K	2 MB	4 sec
50 K	20 MB	33 sec
500 K	197 MB	6 min
5 M	1.7 GB	60 min

particular, we use a customized version of the free tool variety.js for schema extraction on MongoDB collections; we rely on the BIN framework [11] to handle schema mappings and query reformulation (see Appendix); AFD detection is carried out by a simple Javascript algorithm, which determines the presence of AFDs between couples of fields by means of count distinct queries, adopting a smart exploration strategy that reduces the search space like in [6]; finally, OLAP queries are manually formulated. Our reference real-world collection is stored on a single machine (i7 CPU, 32GB RAM) and contains 5M workout sessions with 6 different local schemas (mostly due to missing attributes), 35M exercises and 85M sets. Table 1 shows the execution times for the schema extraction phase. Times are consistent with those of related approaches that perform schema extraction on JSON datasets, such as [1]; also, we note that time increases linearly with the size of the database. Given the low number of schemas, mappings have been manually defined. A sample OLAP query q that groups documents by Facility.Chain (global support 0.38) to obtain the average amount of Exercises.ExCalories (global support 0.69) returns the following indicator values: $sel(q) = 1$ (as there is no filtering), $compl(q) = 0.33$ (lower than the group-by set support), and an average $prec(e)$ of 0.99. The time for executing q on MongoDB is about 3 minutes. Drilling down to Facility.Name (global support 1) increases $compl(q)$ to 1, while the accuracy of the new query with respect to q is 0.98.

As future work, we plan to build a fully-functioning implementation, as well as to thoroughly evaluate the performance and scalability of the approach. Also, we plan to switch from a single machine to a multi-node cluster and to consider schema profiling techniques [9] to enhance the support given to the user at query time.

APPENDIX

Not only inter-schema mappings enable the definition of a global schema, they also allow a MongoDB query formulated on the global schema to be reformulated on each local schema, which is necessary in two situations: (i) when the collection is queried to detect AFDs (Section 5) and (ii) when the user issues an OLAP query on the collection (Section 6). The query reformulation algorithm we adopt here is the one proposed by [11] in the context of *business intelligence networks* (BINs); it enables the rewriting 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 why that algorithm can be reused to safely rewrite queries in both situations (i) and (ii). To this end we need to prove that the data schemas, the inter-schema 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 5.2 can be thought of as a sort of "multi-fact" multidimensional schema with no explicit distinction between levels and measures. However, when an OLAP query is formulated as in Definition 6.1, 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 6.1 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 [11] 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 OLAP query (Definition 6.1) 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 an OLAP 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.

REFERENCES

[1] Mohamed Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Schema Inference for Massive JSON Datasets. In *Proc. EDBT*. Venice, Italy, 222–233.

[2] Carlo Batini, Maurizio Lenzerini, and Shamkant Navathe. 1986. A Comparative Analysis of Methodologies for Database Schema

Integration. *Comput. Surveys* 18, 4 (1986), 323–364.
[3] Philip A Bernstein, Jayant Madhavan, and Erhard Rahm. 2011. Generic schema matching, ten years later. *Proc. VLDB Endowment* 4, 11 (2011), 695–701.
[4] Arnaud Castelltort and Anne Laurent. 2014. NoSQL Graph-based OLAP Analysis. In *Proc. KDIR*. Rome, Italy, 217–224.
[5] Craig Chasseur, Yinan Li, and Jignesh M. Patel. 2013. Enabling JSON Document Stores in Relational Systems. In *Proc. WebDB*. New York, USA, 1–6.
[6] Mohamed Lamine Chouder, Stefano Rizzi, and Rachid Chalal. In press. EXODuS: Exploratory OLAP over Document Stores. *Inf. Syst.* (In press).
[7] Khaled Dehdouh. 2016. Building OLAP Cubes from Columnar NoSQL Data Warehouses. In *Proc. MEDI*. Almería, Spain.
[8] Michael DiScala and Daniel J. Abadi. 2016. Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data. In *Proc. SIGMOD*. San Francisco, USA, 295–310.
[9] Enrico Gallinucci, Matteo Golfarelli, and Stefano Rizzi. 2017. Schema Profiling of Document Stores. In *Proc. SEBD*. Squillace Lido, Italy, 1–8.
[10] Matteo Golfarelli, Simone Graziani, and Stefano Rizzi. 2016. Starry Vault: Automating Multidimensional Modeling from Data Vaults. In *Proc. ADBIS*. 137–151.
[11] Matteo Golfarelli, Federica Mandreoli, Wilma Penzo, Stefano Rizzi, and Elisa Turricchia. 2012. OLAP query reformulation in peer-to-peer data warehousing. *Inf. Syst.* 37, 5 (2012), 393–411.
[12] Matteo Golfarelli and Stefano Rizzi. 2009. *Data warehouse design: Modern principles and methodologies*. McGraw-Hill, Inc.
[13] Rihan Hai, Sandra Geisler, and Christoph Quix. 2016. Constance: An Intelligent Data Lake System. In *Proc. SIGMOD*. San Francisco, USA, 2097–2100.
[14] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *Proc. SIGMOD*. 647–658.
[15] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2013. Discovering implicit schemas in JSON data. In *Proc. ICWE*. 68–83.
[16] Hans-Joachim Lenz and Arie Shoshani. 1997. Summarizability in OLAP and Statistical Data Bases. In *Proc. Ninth International Conference on Scientific and Statistical Database Management*.
[17] Zhen Hua Liu and Dieter Gawlick. 2015. Management of Flexible Schema Data in RDBMSs - Opportunities and Limitations for NoSQL. In *Proc. CIDR*. Asilomar, USA.
[18] Erhard Rahm and Philip A. Bernstein. 2001. A survey of approaches to automatic schema matching. *VLDB J.* 10, 4 (2001).
[19] Oscar Romero and Alberto Abelló. 2006. Multidimensional Design by Examples. In *Proc. DaWaK*. Krakow, Poland, 85–94.
[20] Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. 2015. Inferring Versioned Schemas from NoSQL Databases and Its Applications. In *Proc. ER*. 467–480.
[21] Stefanie Scherzinger, Eduardo Cunha de Almeida, Thomas Cerqueus, Leandro Batista de Almeida, and Pedro Holanda. 2016. Finding and Fixing Type Mismatches in the Evolution of Object-NoSQL Mappings. In *Proc. Workshops EDBT/ICDT*.
[22] William Spoth, Bahareh Sadat Arab, Eric S. Chan, Dieter Gawlick, Adel Ghoneimy, Boris Glavic, Beda Christoph Hammerschmidt, Oliver Kennedy, Seokki Lee, Zhen Hua Liu, Xing Niu, and Ying Yang. 2017. Adaptive Schema Databases. In *Proc. CIDR*. Chaminade, USA.
[23] Boris Vrdoljak, Marko Banek, and Stefano Rizzi. 2003. Designing Web Warehouses from XML Schemas. In *Proc. DaWaK*. 89–98.
[24] Lanjun Wang, Shuo Zhang, Juwei Shi, Limei Jiao, Oktie Hassanzadeh, Jia Zou, and Chen Wangz. 2015. Schema management for document stores. *Proc. VLDB Endowment* 8, 9 (2015), 922–933.