OLAP Querying of Document Stores in the Presence of Schema Variety
(DISCUSSION PAPER)

Matteo Francia, Enrico Gallinucci, Matteo Golfarelli, and Stefano Rizzi

DISI, University of Bologna, Italy

Abstract. Document stores are preferred to relational ones for storing heterogeneous data due to their schemaless nature. However, the absence of a unique schema adds complexity to analytical applications. In a previous paper we have proposed an original approach to OLAP on document stores; its basic idea was to stop fighting against schema variety and welcome it as an inherent source of information wealth in schemaless sources. In this paper we focus on the querying phase, showing how queries can be directly rewritten on a heterogeneous collection in an inclusive way, i.e., also including the concepts present in a subset of documents only.

Keywords: NoSQL · Multidimensional Modeling · OLAP

1 Introduction

Schemaless databases, in particular document stores (DSs) such as MongoDB, 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 a field, and in different structures for instances. 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.

In this paper we propose an approach to OLAP querying on DSs. The basic idea is to welcome data heterogeneity and schema variety as an inherent source of information wealth. So, instead of trying to hide this variety, we show it to users (basically, data scientists and data enthusiasts). To the best of our knowledge, this is the first approach to propose a form of approximated OLAP analyses on document stores that embraces and exploits the inherent variety of documents. OLAP querying is carried out directly on the data source, without

Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0). This volume is published and copyrighted by its editors. SEBD 2020, June 21-24, 2020, Villasimius, Italy.

CEUR-WS.org/Vol-2646/07-paper.pdf

materializing any cube or data warehouse. Remarkably, 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.

2 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. Accessing schemaless sources often requires the adoption of data integration techniques to provide a unified view of data; as this is not the primary focus of the paper, we refer the reader to a survey on the subject [6].

A distinguishing feature of our approach is the definition of a multidimensional representation of the schema to enable OLAP analyses directly on the DS. From this point of view, a work closely related to ours is [2], which proposes a schema-on-read approach for multidimensional queries over DSs. That approach differs from ours because it focuses on the multidimensional representation of JSON data and overlooks the variety issues, and because the detection of functional dependencies is activated on-demand only *after* the user has written a query. OLAP analyses on DSs are enabled also in [7], although a simpler integration approach is proposed, with no identification of functional dependencies nor multidimensional views.

On the issue of schema variety on DSs, a recent work [8] builds on a simple mapping strategy to hide the variety within a single, comprehensive query. Whereas the approach proposes a simpler querying mechanism, it only covers a limited set of schema variants and does not support OLAP.

Finally, several works have focused on bringing NoSQL back to the relational world. [11] discusses an approach to provide schema-on-read capabilities for flexible schema data stored on RDBMSs: it maps the document structure on different tables and provides a *data guide* as the union of every possible field at any level. However, no advanced schema matching mechanism is provided.

3 Approach Overview and Working Example

Figure 2 gives an overview of our approach. Although the picture suggests a sequential execution of the stages, it simply outlines the ordering for the first iteration. 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 based on an example; all the details can be found in [3]. The example we use is based on a real-world collection of workout sessions, WS, obtained from a worldwide company selling fitness equipment. Figure 1 (left) shows a sample document in the collection, organized according to three nesting levels: the first level contains information about the user; the Exercises

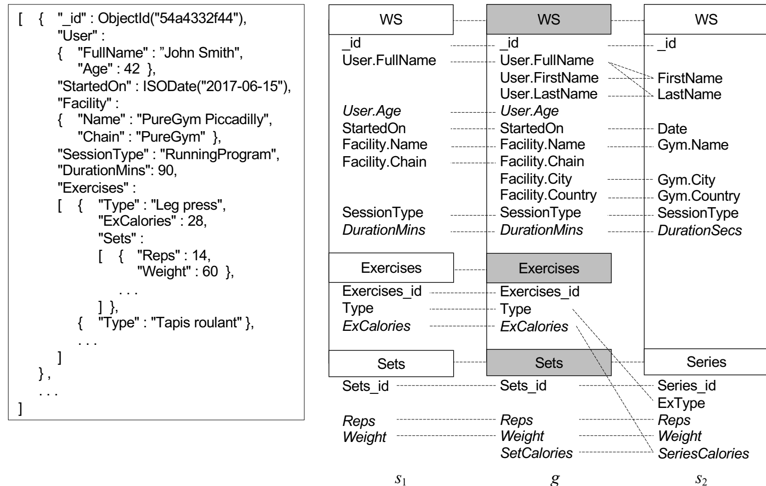

Fig. 1. A JSON document in the WS collection (left), its schema (s_1), another schema of the same collection (s_2), and the global schema (g)

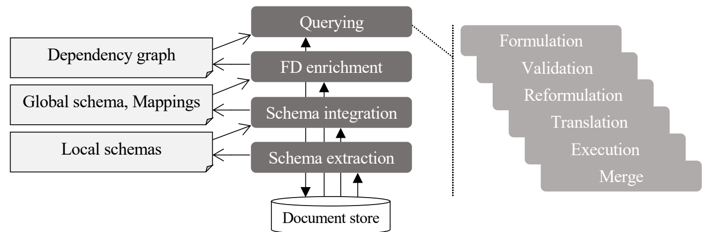

Fig. 2. Approach overview

array contains an object for every exercise carried out during the session; the Sets array contains an object for every set that the exercise was split into.

1. **Schema extraction**. The goal is to identify the set of distinct *local schemas* that occur inside a collection of documents. To this end we adopt a tree-like definition for schemas which models arrays by considering the union of the schemas of their elements, as done in [1]. This is a completely automatic stage; its implementation is loosely inspired by the free tool variety.js and consists of a routine that connects to the desired collection on MongoDB, extracts the local schemas, and writes the results on a triplestore. With reference to our example, Figure 1 shows the schema s_1 of the sample document. 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 nestings of arrays, with the root WS on top.

2. **Schema integration**. The goal here is to integrate the distinct, local schemas extracted from a collection to obtain a single and comprehensive view of the latter, i.e., a *global schema*, and its mappings with each local schema. To this end we rely on both *schema matching* and *schema mapping* techniques: the former allow to identify the single matches between the attributes, while the latter define the mappings between each local schema and the global one, thus enabling the rewriting of queries. A mapping between two (sets of) primitive fields P and P' requires a *transcoding function* to transform values of P into values of P'; these functions enable query reformulation in the presence of selection predicates as well as the integration of the query results obtained from all documents. The approach we adopt for schema integration includes two steps. The first step is automatic and defines a preliminary global schema as the name-based union of all local schemas [10]. 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 [12]) can be used to automatically find a list of possible matches between arrays and primitives; then the user will browse them and possibly define additional mappings. With reference to our example, Figure 1 shows the global schema g resulting from the integration of s_1 with s_2, one more schema from WS; mappings between arrays and primitives are represented with dotted lines. The transcoding functions of mappings $\langle\{\mathsf{Date}\}, \{\mathsf{StartedOn}\}\rangle$ and $\langle\{\mathsf{FirstName}, \mathsf{LastName}\}, \{\mathsf{User.FullName}\}\rangle$ are the identity function and a function that concatenates two strings, respectively.
3. **Schema enrichment**. The goal 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 functional dependencies (FDs) between fields in the global schema. By assuming the presence of identifiers at every nesting level, some exact FDs can be derived from the global schema without looking at the data. However, additional FDs can exist between primitive nodes, though they cannot be inferred from the schema and can only be found by querying the data. More precisely, since DSs may contain incomplete and faulty data, we look for *approximate FDs* (AFDs) [9], i.e., FDs that "mostly" hold on data. To detect approximate FDs, we adapted the approach proposed in [4]. As a result, we determine a *dependency graph*, which provides a multidimensional view of the global schema in terms of the FDs between its primitive fields. Figure 3 shows the dependency graph for our example. Each primitive field f is represented as a circle whose color is representative of the *support* of f, i.e., of the percentage of times that f occurs in the collection (the lighter the tone, the lower the support). Identifiers (e.g., _id) are shown in bold. Directed arrows are representative of the (A)FDs detected during schema enrichment; for instance, we have $\mathsf{_id} \rightarrow \mathsf{Facility.Name}$ (exact FD, in black) and $\mathsf{Facility.Name} \rightsquigarrow \mathsf{Facility.Chain}$ (AFD, in grey). The latter FD is approximate because it only holds for a subset of documents.
4. **Querying**. The last stage enables the formulation of multidimensional queries on the dependency graph and their execution on the collection. First of

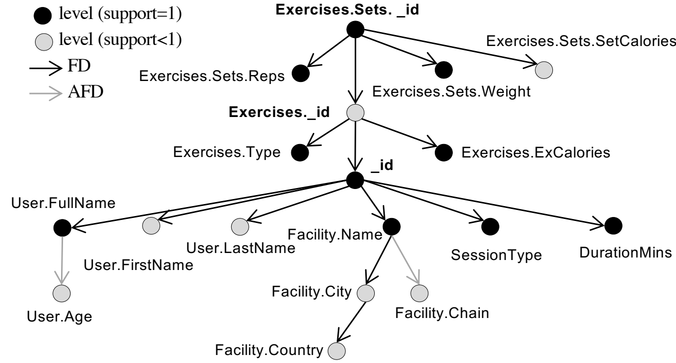

Fig. 3. Dependency graph for the global schema in Figure 1

all, each formulated query is validated against the requirements of well-formedness proposed in the literature [13]. Then, the query is reformulated into multiple queries, one for each local schema in the collection, which are translated into the query language of the DS; the results presented to the user are obtained by merging the results of the single local queries.

4 Querying

In this section we describe the final querying stage of our approach. Given a global schema g and the dependency graph $\mathcal{M}$ obtained by enriching g with (A)FDs, a *multidimensional query* (from now on, md-query) 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 primitive fields in $\mathcal{M}$; p is an optional *selection predicate* defined as a conjunction of Boolean predicates on primitive fields; m is the query *measure*, i.e., the numeric primitive field to be aggregated; φ is the operator to be used for aggregation (e.g., `avg`, `sum`).

For q to be *well-formed*, there must exist in $\mathcal{M}$ one single field $\overline{f}$ such that all the other fields mentioned in q (either in G, p, or m) can be reached in $\mathcal{M}$ from $\overline{f}$. 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.

Other well-formedness constraints for md-queries are introduced in [13]: the *base integrity constraint*, stating that the fields in G must be functionally independent on each other, and the *summarization integrity constraint*. The base integrity constraint can be easily checked on the dependency graph (no arcs must exist between the fields in G). As to the summarization integrity constraint, each query undergoes a check that can possibly return some warnings to inform the user of potentially incorrect results. Specifically, summarization integrity entails *disjointness* (to avoid double counting, the measure instances to be aggregated must be partitioned by the group-by instances) and *completeness* (the union of these partitions must constitute the entire set). Disjointness is easily checked

on the dependency graph by verifying if the granularity of measure m is finer than the one of all the fields in G. Completeness is obtained if all the fields in G have full support, which is easily contradicted in heterogeneous collections. To restore completeness, we adopt at query time the balancing strategies used for incomplete hierarchies in data warehouse design; basically, the `$ifNull` operator in MongoDB is used to replace a missing value in a field with a custom value.

Example 1. 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:

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

We have $fact(q_1) =$ Exercises.Sets._id. Query q_1 passes the validity check with a warning, because the support of Facility.City is less than one, so balancing is used to restore completeness. On the other hand, q_1 meets the disjointness constraint because the granularity Exercises.Sets._id of the required measure, Exercises.Sets.Weight, is finer than both the granularities _id and Exercises._id of the group-by fields. □

Once a well-formed md-query q has been formulated by the user on the dependency graph, it has to be reformulated on each local schema s_i to effectively cope with inter-document variety. To this end we rely on the approach to md-query reformulation proposed in [5] for federated data warehouse architectures. This approach has been proved to be complete and to provide all certain answers to the md-query. As a result, a set of local queries $q^{(i)}$, one for each local schema s_i, are determined.

At this point, each local query $q^{(i)}$ is separately executed on the DS; specifically, $q^{(i)}$ must target only the documents that belong to local schema s_i. 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 WS-schemas in our example) in the following form: a document is created for every schema s_i, containing an array ids with the _id of every document having schema s_i. Then, query $q^{(i)}$ is executed by joining it with the list of identifiers in WS-schemas. Note that, to be executed, $q^{(i)}$ needs be translated to the MongoDB query language, which allows us to declare a multi-stage pipeline of transformations to be carried out on the documents of a collection.

Finally, a post-processing activity is required to integrate and, possibly, further aggregating the results coming from the different local queries. This operation can be performed in-memory, as OLAP queries usually produce a limited number of records and the transcoding functions provide homogeneous values.

Example 2. Consider an md-query that calculates the total amount of burnt calories by facility, excluding workout sessions that are shorter than 30 minutes:

$$q_2 = \langle\{\mathsf{Facility.Name}\}, (\mathsf{DurationMins} \geq 30), \mathsf{Exercises.ExCalories}, \mathtt{sum}\rangle$$

Consider the local schemas s_1 and s_2 in Figure 1. The reformulation of q_2 onto s_1 has no effect (i.e., $q_2^{(1)} \equiv q_2$); conversely, the reformulation onto s_2 generates

the following local query:

$$q_2^{(2)} = \langle\{\mathsf{Gym.Name}\}, (\frac{\mathsf{DurationSecs}}{60} \geq 30), \mathsf{Series.SeriesCalories}, \mathtt{sum}\rangle$$

The MongoDB query obtained from q_1 of Example 1 is the following; note that the missing values of Facility.City are replaced by those of Facility.Name:

```
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] } } } } } )
```

5 Conclusions and Evaluation

In this paper we have presented an original approach to OLAP querying on DSs. Our basic claim is that the heterogeneity and schema variety intrinsic to DSs should be considered as a source of information wealth. 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.

As a proof of concept for our approach we have developed a Java prototype to support the main phases and tested it on a cluster of seven CentOS 6 machines with an 8-core i7-4790 CPU @3.60 GHz and 32 GB of RAM. Our reference real-world collection, WS, is stored on 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.

Query formulation and translation to MongoDB are done in negligible time. Reformulation is done with polynomial complexity [5]. Thus, performances mainly depend on query execution. Consider for instance the following three md-queries:

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

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

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

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,

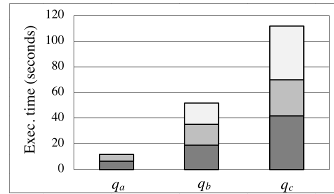

Fig. 4. Execution times of three queries (q_a, q_b, and q_c) split into the execution times of the required local queries

to merge the local queries that involve the same fields on different local schemas; for instance, with reference to Figure 1, 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 are shown in Figure 4; the times for q_b and q_c are higher due to the necessity of unwinding arrays.

References

1. Baazizi, M.A., Lahmar, H.B., Colazzo, D., Ghelli, G., Sartiani, C.: Schema inference for massive JSON datasets. In: Proc. EDBT. pp. 222–233 (2017)
2. Chouder, M.L., Rizzi, S., Chalal, R.: EXODuS: Exploratory OLAP over document stores. Inf. Syst. **79**, 44–57 (2019)
3. Gallinucci, E., Golfarelli, M., Rizzi, S.: Approximate OLAP of document-oriented databases: A variety-aware approach. Inf. Syst. **85**, 114–130 (2019)
4. Golfarelli, M., Graziani, S., Rizzi, S.: Starry vault: Automating multidimensional modeling from data vaults. In: Proc. ADBIS. pp. 137–151 (2016)
5. Golfarelli, M., Mandreoli, F., Penzo, W., Rizzi, S., Turricchia, E.: OLAP query reformulation in peer-to-peer data warehousing. Inf. Syst. **37**(5), 393–411 (2012)
6. Golshan, B., Halevy, A.Y., Mihaila, G.A., Tan, W.: Data integration: After the teenage years. In: Proc. PODS. pp. 101–106 (2017)
7. Hamadou, H.B., Gallinucci, E., Golfarelli, M.: Answering GPSJ queries in a polystore: A dataspace-based approach. In: Proc. ER. pp. 189–203 (2019)
8. Hamadou, H.B., Ghozzi, F., Péninou, A., Teste, O.: Towards schema-independent querying on document data stores. In: Proc. DOLAP (2018)
9. Ilyas, I.F., Markl, V., Haas, P., Brown, P., Aboulnaga, A.: CORDS: Automatic discovery of correlations and soft functional dependencies. In: Proc. SIGMOD. pp. 647–658 (2004)
10. Klettke, M., Störl, U., Scherzinger, S., Regensburg, O.: Schema extraction and structural outlier detection for JSON-based NoSQL data stores. In: Proc. BTW. vol. 2105, pp. 425–444 (2015)
11. Liu, Z.H., Gawlick, D.: Management of flexible schema data in RDBMSs - opportunities and limitations for NoSQL. In: Proc. CIDR. Asilomar, USA (2015)
12. Maßmann, S., Raunich, S., Aumüller, D., Arnold, P., Rahm, E.: Evolution of the COMA match system. In: Proc. OMISWC. Bonn, Germany (2011)
13. Romero, O., Abelló, A.: Multidimensional design by examples. In: Proc. DaWaK. pp. 85–94. Krakow, Poland (2006)