View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Open Archive Toulouse Archive Ouverte

Open Archive Toulouse Archive Ouverte

OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible

This is an author's version published in: http://oatao.univ-toulouse.fr/26237

Official URL
https://doi.org/10.1007/978-3-030-33223-5_16

To cite this version: Ben Hamadou, Hamdi and Gallinucci, Enrico and Golfarelli, Matteo *Answering GPSJ Queries in a Polystore: a Dataspace-Based Approach.* (2019) In: 38th International Conference on Conceptual Modeling (ER 2019), 4 November 2019 (Salvador de Bahia, Brazil).

Any correspondence concerning this service should be sent to the repository administrator: tech-oatao@listes-diff.inp-toulouse.fr

Answering GPSJ Queries in a Polystore: A Dataspace-Based Approach

Hamdi Ben Hamadou[1], Enrico Gallinucci[2], and Matteo Golfarelli[2(✉)]

[1] Institut de Recherche en Informatique de Toulouse, Toulouse, France
hamdi.ben-hamadou@irit.fr
[2] University of Bologna, Cesena, Italy
{enrico.gallinucci,matteo.golfarelli}@unibo.it

Abstract. The discipline of data science is steering analysts away from traditional data warehousing and towards a more flexible and lightweight approach to data analysis. The idea is to perform OLAP analyses in a *pay-as-you-go* manner across heterogeneous schemas and data models, where the integration is progressively carried out by the user as the available data is explored. In this paper, we propose an approach to support data analysis within a polystore supporting relational, document and column data models by automatically handling both data model and schema heterogeneity through a dataspace layer on top of the underlying databases. The expressiveness we enable corresponds to GPSJ queries, which are the most common class of queries in OLAP applications. We rely on Nested Relational Algebra to define a cross-database execution plan. The plan is composed of several local plans, to be executed on the distinct databases, and a global plan, which combines and possibly aggregates inter-database data. The system has been prototyped on Apache Spark.

Keywords: Polystore · NoSQL · Dataspace · GPSJ · Schemaless · OLAP

1 Introduction

With the rise of Big Data, NoSQL systems have effectively provided different ways to address the scalability issues of relational database management systems (RDBMSs) and the variety aspect of Big Data. As companies move towards *polyglot persistence* [20] (i.e., employing several DBMSs to exploit the best features of each) to optimize the operational workload, new challenges arise from an analytical perspective, because the analyst needs a transparent way to access these fragmented and differently-shaped data. At the same time, the discipline of data science is steering analysts away from traditional data warehousing and towards a more flexible and lightweight data analysis approach. The idea is to relax the rigidity of traditional integration approaches to perform OLAP (OnLine Analytical Processing) analyses in a *pay-as-you-go* manner [14], where the integration is

https://doi.org/10.1007/978-3-030-33223-5_16

progressively carried out by the user as the available data is explored. This calls for new approaches to enable effective analyses on a polyglot system without performing a complex integration phase.

The main challenges to address in this context are related to the heterogeneity of the data in terms of data model and schema. *Data model heterogeneity* is intrinsic in a polyglot database; it requires to distribute the computation of a query across the different databases (which adopt different query languages) and to possibly rely on a middleware to combine and further elaborate the results. *Schema heterogeneity* is a common type of heterogeneity in most NoSQL systems as they abandon the traditional *schema-first, data-later* approach of RDBMS (which requires all record in a table to comply with a predefined schema) in favour of a *soft-schema* approach, in which each record embeds its own schema definition. For instance, two records in the same collection may contain different attributes or the same attributes following different naming conventions. Schema heterogeneity is mainly due to schema evolution and to the acquisition of data from sources adopting different schema representations for the same entities.

State-of-the-art proposals for polyglot systems mainly include *multistores* (which provide a unique query language to separately query different DBMSs) and *polystores* (which additionally enable cross-DBMS query processing) [23]. Current solutions mostly focus on addressing data model heterogeneity and on optimising the query processing, but they do not consider schema heterogeneity. This prevents analysts from taking full advantage of the data, as several instances may be missed by queries that do not take schema variations into consideration. In this paper we propose an approach to support data analysis within a polystore by handling both data model and schema heterogeneity through a dataspace layer on top of the underlying databases. A dataspace is a lightweight integration approach providing basic query expressive power on a variety of data sources, bypassing the complexity of traditional integration approaches and possibly returning best-effort or approximate answers [7]. Consistently with the pay-as-you-go philosophy, the dataspace is first built by applying simple matching rules and is progressively enriched by the users as they discover new relationships among data structures through exploratory queries.

The query expressiveness we enable corresponds to GPSJ queries (i.e., generalized projection, selection and join [12]), i.e., the most common class of queries in OLAP applications. State-of-the-art works typically delegate to the user the formulation of adequate queries with the risk of getting inconsistent answers to the envisioned questions. In contrast, GPSJs enforce a query semantics to prevent the user from getting misleading results leading to ambiguous or potentially incorrect interpretation in the analytical context. The possibility to extend the approach to a broader class of queries is considered as future work. For a given GPSJ, our approach defines a cross-database execution plan in Nested Relational Algebra (NRA) [24], which is compatible with the expressiveness of document stores' query language [3] and SQL (as it is a superset of relational algebra), with the latter being used by both RDBMSs and column-based systems. The cross-database execution plan is composed of several local plans, to be executed

on the distinct underlying databases, and a global plan, which combines and possibly aggregates inter-database data. The resolution of schema heterogeneity is handled in the local plans, where the knowledge of the dataspace is exploited to properly query all schema variations of the involved data. This activity is supported by previous research efforts on enabling schema-independent querying on heterogeneous schemas [1,2,9,10], which focus only on single collections of records in a particular data model. A prototypical implementation of the approach has been carried out on Apache Spark [26].

The paper outline is as follows. After discussing related work in Sect. 2, in Sect. 3 we formalize the dataspace and the query expressiveness. Then we present the formulation of the execution plan in Sect. 4. Finally, in Sect. 5 we briefly discuss the prototypical implementation and we draw the conclusions.

2 Related Literature

The importance of transparently querying multistore systems has been highlighted by contexts such as federated databases [21] and, more recently, soft-schema support in NoSQL systems [5]. Here we classify state-of-the-art work by focusing on the considered levels of data model and schema heterogeneity.

Data Model Transformation. Generally, these works store document data model into a relational one [6,22]. They offer relational views built on top of the new relational data model to assist the user while formulating queries. This strategy implies that several data model transformation should be performed. Hence, this process requires additional resources, such as an external relational database [15]. Users of these systems have to learn new schemas every time new data are inserted (or updated) in the collection, because it is necessary to re-generate the relational views.

Multistore and Polystores. Most of the approaches provide integrated access to a number of heterogeneous database systems [8,16] through one [16] or more query language [8] using a middle-ware layer. However, they still require the user to either define the global schema or to specify a particular data source to use, e.g., BigDAWG [8] requires user to use the adequate querying language for each data model. Furthermore, they consider neither schema mapping during the query rewriting steps, nor schema heterogeneity.

Multimodel Systems. These systems offer a single platform to store and query data in different data models (e.g., OrientDB, http://orientdb.com/orientdb/). Multimodel systems excel in term of data governance, management, and access. However, they are limited to a pre-defined set of data models and extending support to new data models is challenging.

Schema-Independent Querying. In document-based stores *structural heterogeneity* points to the existence of several paths to access the same attribute. A transparent querying mechanisms to overcome this heterogeneity is introduced in [2]. A recent research work [9] resolves the problem of having semantically equivalent

attributes but with a *different naming convention*, as highlighted in [25], using a set of schema mappings. Most of these approaches consider the heterogeneity problem inside one collection at a time for a particular data model only. Moreover, the same information could be represented using *different data types*, and transcoding functions are required to resolve this heterogeneity [11].

Schema Inference. A second line of work focuses on the representation of the different schemas within the same collection of documents. In [25] the authors recommend summarizing all document schemas under a skeleton to discover the existence of fields or sub-schemas inside the collection. In [13] the authors suggest extracting collection structures to help developers in the process of designing their applications. The limitation with such a logical view is that it requires a manual process in order to build the desired queries by including the desired attributes and all their possible navigational paths.

All mentioned works handle either data model or schema heterogeneity. To the best of our knowledge, this is the first work to handle both of them.

3 Dataspace and Query Modeling

In this work we consider a polystore comprising databases in three data models: relational, document-based and column-based[1]. Our running example is a variation of Unibench [18], i.e., a benchmark multimodel dataset based on an e-commerce application. The conceptual schema is shown in Fig. 1. With respect to Unibench we exclude the graph and key-value databases and we extend the benchmark by injecting some heterogeneity into the schemas. We remark that schema heterogeneity is possible only in the document-based and column-based data models. In particular, we cover the following kinds of schema heterogeneity.

- Missing attributes: attributes that exist in some records and not in others (e.g., the gender and birthday of the Client are not always specified).
- Different data types: attributes with varying data types (e.g., the id in Client is a number, but the personId in Order is stored as string).
- Semantic equivalence: attributes with varying naming conventions (e.g., orderLine.cost and orderLine.price in Order are alternative attributes representing the same information).

In the polystore, the data is split among a set of databases DB. We exploit the concept of dataspace to provide a global representation of the available attributes in the different databases and to hide the underlying schema heterogeneity. In particular, the dataspace plays the role of the abstraction level enabling the user to formulate queries. As data model heterogeneity entails terminology heterogeneity, Table 1 explains the terminology used in the remainder of the paper to generally refer to schema elements (e.g., tables, columns), independently of their declination in the different data models. The basic information we consider is the *attribute*, which we define as follows.

[1] We remark that column-based NoSQL systems (e.g., BigTable [4]) are different from column-oriented DBMS (e.g., Vertica).

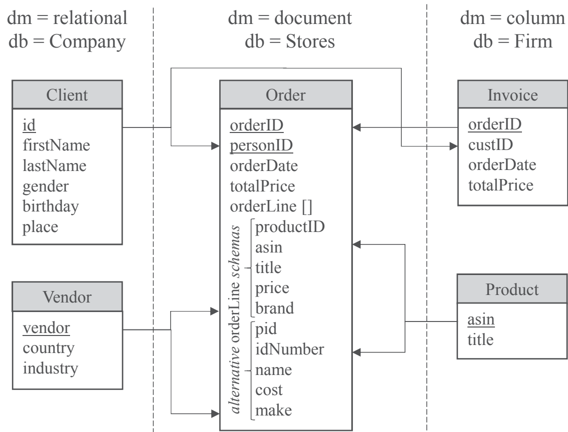

Fig. 1. Running example of a multi-store e-commerce application, based on Unibench [18]; orderLine is an array in the Order collection whose objects come in two schema variations (i.e., attributes from orderLine.productID to orderLine.brand are alternative to those from orderLine.pid to orderLine.make).

Table 1. The adopted terminology VS the terminology used in different data models.

Relational	Column-based	Document-based	Reference term
Table	Column-family	Collection	Collection
Tuple	Object	Document	Record
Attribute	Column	Attribute	Attribute
Attribute name	Column name	Path	Name

Definition 1 (Attribute). *Given a polystore DB, we define an attribute as* $a = (dm, db, col, name)$*, where* $dm =$ [relational|column|document] *is the data model,* $db \in DB$ *is the database name,* col *is the collection name in* db*,* $name$ *is the name of the attribute in the collection* col*.*

We refer to A^* as the set of all attributes within all databases in the polystore DB; given an attribute a, we use $db(a)$, $col(a)$, $name(a)$ and $array(a)$ to respectively refer to its database db, its collection col, its name $name$ and (possibly) the *array* attribute in which it is contained. In fact, attributes in document-based stores can appear in a *nested* form. In such cases, the name of the attribute corresponds to a path in dot notation that contains the ordered list of *array* attributes and ends with the attribute itself; accessing a simple attribute requires to *unnest* all the arrays in which it is contained.

Example 1. With respect to Fig. 1, consider the following reference attributes:

- a_1 : (relational, Company, Client, id)
- a_2 : (relational, Company, Client, firstName)
- a_3 : (relational, Company, Client, lastName)
- a_4 : (document, Stores, Order, personID)
- a_5 : (document, Stores, Order, orderLine)
- a_6 : (document, Stores, Order, orderLine.productID)
- a_7 : (document, Stores, Order, orderLine.pid)
- a_8 : (document, Stores, Order, orderLine.price)
- a_9 : (document, Stores, Order, orderLine.cost)
- a_{10} : (document, Stores, Order, orderLine.brand)
- a_{11} : (document, Stores, Order, orderLine.make)

It is $db(a_4)$ = Stores and $col(a_4)$ = Order; also, $array(a_6) = array(a_7) = array(a_8) = array(a_9) = a_5$; attributes a_6, a_8 and a_{10} belong to the first schema variation of orderLine, while attributes a_7, a_9 and a_{11} belong to the second schema variation of orderLine.

In a polystore, attributes do not provide a global representation that hides the inherent schema heterogeneity as several syntactically different attributes may represent the same type of information. Relationships between attributes can be either manually inserted or automatically discovered. The automatic retrieval of such relationships is out of scope in this paper. Nonetheless, the literature on this topic is abundant; we refer the reader to a survey on common techniques for schema matching [17] and to an existing work for automatic discovery of primary-foreign key relationships [19]. Whether they are obtained either automatically or manually, which is likely when an incremental approach is adopted, relationships can be formalized as follows:

Definition 2 (Mapping). *A mapping is a relationship between two attributes $a^{'}$ and $a^{''}$. We define a mapping as $m = (a^{'}, a^{''}, \phi, \varphi, \psi)$, where $a^{'}, a^{''} \in A^*$, $\phi = [\mathsf{sameAs}|\mathsf{fk}]$ is the type of the mapping, and φ is a transcoding function to express a' values in a'' format (if necessary; otherwise, $\varphi = I()$ where $I()$ is the identity function). Finally, ψ is the semantics describing the meaning of the relationship (limitedly to* fk *mappings).*

The mapping type sameAs resolves semantic equivalence by indicating that there is an exact match between $a^{'}$ and $a^{''}$, i.e., both attributes represent the same information for a given entity; a sameAs mapping can exist only if, for any given record, $a^{'}$ and $a^{''}$ never coexist. Conversely, fk indicates that the values in $a^{'}$ correspond to the values in $a^{''}$ (i.e., a relationship that, in RDBMSs, is modeled as $a^{'}$ being a *foreign key* to $a^{''}$). Consequently, $a^{''}$ must be a key; for the sake of simplicity, all keys are not composite. Mappings are assumed to be consistent; for example if $\exists\ m^{'} = (a^{'}, a^{''}, \mathsf{fk}, \varphi, \psi)$, then $\nexists\ m^{''} = (a^{'}, a^{''}, \mathsf{sameAs}, \varphi')$.

The sameAs mappings are used to capture schema heterogeneity within a collection (thus, $db(a^{'}) = db(a^{''})$ and $col(a^{'}) = col(a^{''})$) whereas fk mappings are used to establish join relationships between collections (thus, $col(a^{'}) \neq col(a^{''})$). The semantics is necessary when the same attribute is referenced by several fk mappings to disambiguate the relationships. Note that while fk mappings are oriented, sameAs mappings are not oriented in principle, but they become oriented in practice when we consider the function φ that transcodes from a' to a'' and not viceversa.

Example 2. Consider the following mappings between the attributes defined in Example 1: $m_1 = (a_4, a_1, \mathsf{fk}, toInt(), \text{“}client\ order\text{”})$, $m_2 = (a_6, a_7,\ \mathsf{sameAs}, I())$, $m_3 = (a_8, a_9,\ \mathsf{sameAs}, I())$, $m_4 = (a_{10}, a_{11},\ \mathsf{sameAs}, I())$.

The presence of several attributes that semantically represent the same concept can be hidden by an abstract representation called *feature*, which is based on the sameAs mappings.

Definition 3 (Feature). *A feature is a representation of a set of attributes in the polystore that semantically model the same concept. We define a feature as* $f = (name, a, M)$*, where* a *is the* representative *attribute of the feature, name is the name of the feature (possibly different from* $name(a)$*), and* M *is a set of* sameAs *mappings, in the form* $(a', a,$ sameAs$, \varphi)$*, linking all the feature's attributes to the representative attribute* a*.* $M = \varnothing$ *when a concept is modeled by a single attribute.*

The name of each feature is derived from the names of the represented attributes. However, it is up to the end user to specify a different name.

Example 3. Given the mappings in Example 2 we obtain the following features:

- $f_1 =$ (id, a_1, $\varnothing$)
- $f_2 =$ (firstName, a_2, $\varnothing$)
- $f_3 =$ (lastName, a_3, $\varnothing$)
- $f_4 =$ (personId, a_4, $\varnothing$)
- $f_5 =$ (orderLine, a_5, $\varnothing$)
- $f_6 =$ (orderLine.productID, a_6, $\{m_2\}$)
- $f_7 =$ (orderLine.price, a_8, $\{m_3\}$)
- $f_8 =$ (orderLine.brand, a_{10}, $\{m_4\}$)

We refer to $attr(f)$ as the set of attributes represented by f (i.e., the representative attribute plus those derived from the mappings). An attribute is always represented by one and only one feature; thus, for any two features $f^{'}$ and $f^{''}$, it is $attr(f^{'}) \cap attr(f^{''}) = \varnothing$. We refer to the feature of an attribute a as $feat(a)$ and to the name of a feature as *name(f)*.

Ultimately, we simply define the dataspace as follows.

Definition 4 (Dataspace). *A dataspace* $\mathcal{D}$ *is a set of features.*

We remark that, since features represent only attributes, there is no notion of collection in the dataspace (i.e., at the feature level). This is a substantial difference with a traditional integration approach, which would have required to define global collections and to model them (and their respective attributes) consistently with the modelings used in the different databases. Instead, features simply highlight the semantically distinct concepts that are available in the dataspace. In the next Section we explain the query mechanism based on the dataspace of features.

The query expressiveness that we consider covers a wide class of queries by composing three basic SQL operators: selection, join and generalized projection. The combination of these three operators determines GPSJ (Generalized Projection / Selection / Join) queries that were first studied in [12]. We provide the following definition of a query, which is based on the features of the dataspace.

Definition 5 (Query). *Given a dataspace* $\mathcal{D}$*, we define a query as* $q = (q_\pi, q_\gamma, q_\sigma)$*, where:* $q_\pi \subseteq \mathcal{D}$ *specifies the features to be projected;* q_γ *specifies optional aggregations as a set of couples* (f, op)*, where* $f \in \mathcal{D}$ *and op is an aggregation function;* q_σ *is an optional set of selection predicates in the form of triplets* (f, ω, v)*, where* $f \in \mathcal{D}$*,* $\omega \in \{=; >; <; \neq; \geq; \leq\}$ *and* v *is a value.*

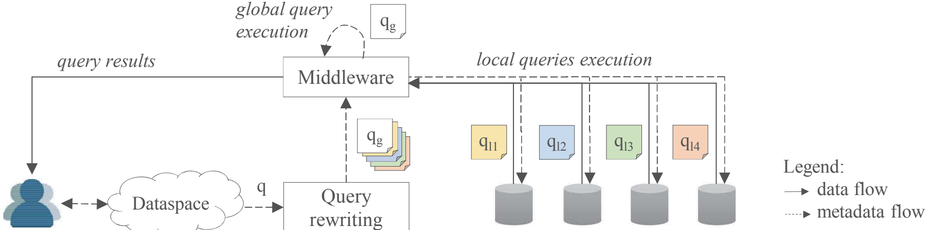

Fig. 2. Query execution process: the query q (formulated on the dataspace) is translated into a set of local queries ($q_{l1} \dots q_{l4}$) to be executed in separate databases, and a global query (q_g) that operates in the middleware on the results of the local queries.

GPSJ queries are the most common class of queries in OLAP applications. Attributes in q_γ are measures of the event that is the target of the OLAP analysis. The single events are measured at the finest level of granularity, possibly filtered by conditions expressed on q_σ and then grouped at the coarser granularity defined by q_π. It is not mandatory that all the three operators are present, thus simple selection queries and join queries are also covered. We refer to $feat(q)$ as the set of features involved in q; also, we will use $attr(q)$ as short for $attr(feat(q))$.

Example 4. Let us suppose that we want to measure the average price orderLine.price of the products orderLine.productID of brand orderLine.brand "ABC" by a client called "John Smith" from the dataspace $\mathcal{D}$. Therefore the group-by set is $q_\pi = \{f_6\}$; the aggregation set is $q_\gamma = \{(f_7, \mathsf{avg})\}$ and the set of selection predicates is $q_\sigma = \{(f_2, =, \text{"}John\text{"}), (f_3, =, \text{"}Smith\text{"}), (f_8, =, \text{"}ABC\text{"})\}$.

4 Execution Plan Formulation

The execution of the query requires the definition of an execution plan that potentially includes different databases. We model the execution plan in NRA, as it is compatible with SQL and document stores' query languages [3]. Given a query execution plan, we distinguish between the single *local plans* (i.e., the parts that can be executed directly on a single database) and the *global plan* (i.e., the part to be executed in the middleware to join the data coming from different databases). While the local plans directly access the collections of the polystore, the global plan accesses the intermediary results of the local plans (i.e., *views* on the single databases). An intuition of the process is given in Fig. 2. We remark that schema variability is managed by the local plans.

4.1 Determining the Query Graph

The information necessary to build the query plan can be modeled by means of a supporting structure we call *datagraph*. Indeed, a query involves a set of features which, in turn, represent several attributes in the dataspace. The datagraph is used to find the connections between these attributes and to obtain the execution plan for a given query.

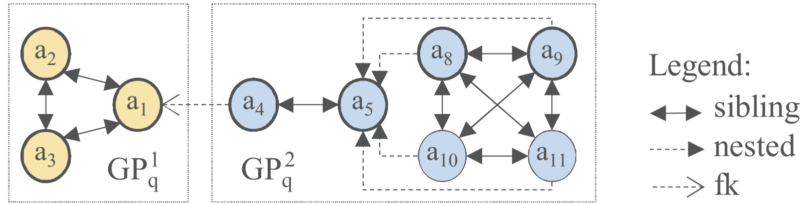

Fig. 3. Query graph of the query in Example 4, with the query graph partitions GP_q^1 and GP_q^2 highlighted; bold circles correspond to representative attributes of a feature.

Definition 6 (Datagraph). *The datagraph G is a graph $G = (A^*, E)$ where A^* is the set of all the attributes of all databases (representing the vertexes of the graph) and E is the set of edges connecting the attributes.*

An edge $e \in E$ between two attributes $a^{'}$ and $a^{''}$ indicates the existence of a relationship, which is described by its type, i.e., *type*(e); its value is one of the following three:

- sibling: represented as $a^{'} \leftrightarrow a^{''}$, it indicates that $a^{'}$ and $a^{''}$ are in the same collection and at the same nesting level;
- nested: represented as $a^{'} \xrightarrow{\text{n}} a^{''}$, it indicates that $a^{'}$ is nested inside $a^{''}$;
- fk: represented as $a^{'} \xrightarrow{\text{fk}} a^{''}$, it indicates that the values of $a^{''}$ are referred to the values of $a^{'}$.

Edges of type sibling and nested are automatically derived from the schema, while those of type fk can be either derived from the original schemas or defined by the user through mappings. Noticeably, nested edges can only come from databases whose data model supports nested attributes (i.e., document- and column-based). Figure 3 shows a portion of the datagraph representing the attributes from Example 1. The existence of a directed path from $a^{'}$ to $a^{''}$, represented as $a^{'} \Rightarrow a^{''}$, implies the existence of a *-to-one* (i.e., either *one-to-one* or *many-to-one*) relationship from $a^{'}$ to $a^{''}$ through a chain of join and unnesting operations. For instance, it is $a_9 \Rightarrow a_2$, while $a_1 \not\Rightarrow a_4$.

Definition 7 (Query graph). *Given a datagraph G and a query q, we define the query graph $G_q = (A' \subseteq A^*, E' \subseteq E)$ as the minimally connected subgraph of G such that* i) *$A' \supseteq attr(q)$, and* ii) *there exists $A'' \subseteq A'$ s.t. $A'' \neq \varnothing, A'' \supseteq q_\gamma, \forall (a \in A'', a' \in A')$, it is $a \Rightarrow a^{'}$.*

Condition (i) ensures that all attributes belonging to the features involved in the query are included in A'. Condition (ii) entails the *answerability* of query q on D with the GPSJ semantics, that is, there exist one or more attributes representing the events at the finest level of granularity (i.e., a *-to-one* relationship exists with all the others attributes in q). More than one query graphs could exist for a given query as more than one *-to-one* paths could exist each associated to a different semantics (e.g., a sale could be associated to both the *date of sale* and *date of shipping*). In this case the user is asked to identify the adequate query graph to execute.

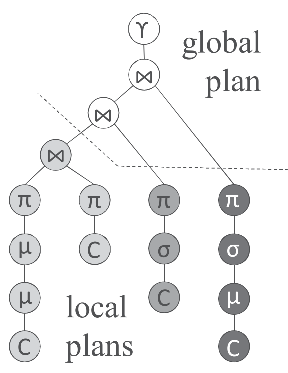

Fig. 4. Sample execution plan for a generic GPSJ; different shades of grey represent different databases.

Algorithm 1. Definition of the NRA execution plan for a query q.

Input $q = (q_\pi, q_\gamma, q_\sigma)$: a query; $G_q = (A', E')$ the query graph for q.
Output P: the NRA plan of q.
1: $P \leftarrow \varnothing$
2: $LP \leftarrow \varnothing$ ▷ Empty array of local plans
3: $GP_q \leftarrow partitionQueryGraph(G_q)$
4: **for all** $GP_q^i \in GP_q$ **do** ▷ One local plan is created for every partition of G_q
5: $CP \leftarrow \varnothing$ ▷ Empty array of collection plans
6: $C \leftarrow identifyAccessedCollections(GP_q^i)$
7: **for all** $col \in C$ **do** ▷ One collection plans is created for every partition of GP_q^i
8: $CP_{col} \leftarrow defineCollectionPlan(col, GP_q^i)$
9: $LP_i \leftarrow defineLocalJoins(CP, GP_q^i)$
10: $P \leftarrow defineGlobalPlan(LP, G_q)$
11: **return** P

4.2 Defining the Nested Relational Algebra Execution Plan

The full structure of a GPSJ query is shown in Fig. 4 and, as discussed in Sect. 3, is composed of an aggregation[2], over a set of joins, over a set of filtering operators. The process to translate a query graph G_q into an NRA execution plan is described by Algorithm 1 and requires to: (1) partition G_q in several subgraphs, each corresponding to a local plan (line 1); (2) define each local plan (lines 4–9); (3) collate the local plans into the global one (line 3).

Query Graph Partitioning. Intuitively, a local plan includes all and only the operators that apply to the same database. More formally, this corresponds to partitioning G_q based on the edges of type fk in E' (denoted as E'_{glo}) such that $a' \xrightarrow{\mathsf{fk}} a''$ and $db(a') \neq db(a'')$ (see Fig. 3). Let us define GP_q as the set of partitions, where $|GP_q| = |E'_{\mathsf{glo}}| + 1$. Noticeably, if two edges in E'_{glo} refer to the same database db, it will determine two local plans. For instance, with reference to the running example, this happens if both Client and Vendor tables are accessed on the relational database through the Stores collection in the document database.

[2] We define the aggregation with the operator γ declared as ${}_X\gamma_Y$, where X is the group-by set (i.e., a set of features) and Y is the set of aggregations (where each aggregation is composed of a feature and an aggregation function).

Local Plan Definition. At this point, for each query graph partition GP_q^i, we define the corresponding local plan by applying in sequence the following steps.

1. *Identify accessed collection* Similarly to the query graph partitioning step, the collections to be accessed are identified by partitioning G_q^i based on the edges of type fk. It is possible that the same collection needs to be accessed twice (e.g., given a collection of cities, both the birth city and the residence of customers are requested by the query); this happens when G_q^i includes two fk edges between the same collections and with different semantics.
2. *Define collection plan* For each collection *col* we define a plan by applying in sequence the following steps.
   (a) *Collection accesses* A collection access $\mathsf{C}(col)$ is added to the local plan to denote the collection to be accessed.
   (b) *Unnest operators* Given a feature $f \in feat(q)$, it may happen that some of the $attr(f)$ belong to a nested structure. To retrieve them it is mandatory to flatten the structure by recursively unnesting the arrays. More formally, if $\exists\ a \in attr(f) \mid array(a) \neq \varnothing$, the unnest operator μ on $array(a)$ is necessary. For instance, given $a^{'''} \in attr(f)$ in the collection col, if $array(a^{'''}) = a^{''}$, $array(a^{''}) = a^{'}$ and $array(a^{'}) = \varnothing$, then $\mathsf{C}(col)$ in the local plan becomes $\mu_{a''}(\mu_{a'}(\mathsf{C}(col)))$. Notice that, due to schema heterogeneity, several arrays may need to be unnested, thus the unnesting rule is applied to each $a \in attr(f)$.
   (c) *Selection operators* for each feature $f \in feat(q_\sigma)$, a selection operator σ_p must be added to the local plan, where $p = (f, \omega, v)$ is the selection predicate on f. Clearly, p must be actually formulated on $attr(f)$; however, if $|attr(f)| > 1$ due to schema heterogeneity, the same predicate must be applied to several attributes. The predicate must be true for any of the schema variations of f. Each record fits a specific schema variation including only one of the attributes in $attr(f)$, thus p is defined as a disjunction of conditions on $attr(f)$: $p = (\bigvee_{\forall a_i \in attr(f)} \varphi_{a_i}(name(a_i)), \omega, v)$, where φ is the function transcoding a_i into the representative attribute of f. For the sake of optimization, a single selection operation is generated for predicates that must be applied to the same collection, e.g., given $p_1 = \{f^{'}, \omega_1, v_1\}$ and $p_2 = \{f^{''}, \omega_2, v_2\}$, if $col(attr(f^{'})) = col(attr(f^{''}))$ then the applied selection operator is $\sigma_{p_1 \wedge p_2}$.
   (d) *Projection operators* The role of projection operator is threefold: (1) it keeps only the features required by the following join and aggregation operators; (2) it solves the semantic equivalence by combining all the attributes in $attr(f)$ and renaming them in $name(f)$; (3) it solves data format heterogeneity by applying φ to transcode values from the original format to the one of the representative attribute. Consider $F_\pi = \{feat(q_\pi) \ \cup \ feat(q_\gamma) \ \cup \ F_{\bowtie}\}$ the set of features to be projected, where $F_{\bowtie}$ is the set of features whose attributes are involved in fk edges in G_q. Also, consider $F_\pi^{col} = \{f \in F_\pi \mid attr(f) \in col\}$. $|F_\pi^{col}|$ projections are added to the previously defined access plan for *col*. The projection for $f \in F_\pi^{col}$ is defined as $\left(\bigvee_{\forall a_i \in attr(f)} \varphi_{a_i}(name(a_i))\right) \ / \ name(f)$. The role

of $\bigvee$ is to select the only non-null value among $attr(f)$; it is expressed with the CASE statement in SQL, or with the $ifNull operator in the MongoDB query language. Finally, "/" represents the renaming of the result with the feature's name.

3. *Define local joins* For each edge $a' \xrightarrow{\text{fk}} a''$ in a query graph partition G_q^i, a join operator $\bowtie_{name(feat(a'))=name(feat(a''))}$ is added to join the different collection plans. Please note that for the sake of simplicity we did not consider projections aimed at removing features that are necessary only for joins.

Global Plan Definition. Similarly to the addition of local joins, a join operator $\bowtie_{name(feat(a'))=name(feat(a''))}$ is added for each edge $a' \xrightarrow{\text{fk}} a''$ between two query graph partitions to join the different local plans. We remark that the optimization of join ordering is out of the scope of this paper. Ultimately, the aggregation operator ${}_{q_\pi}\gamma_{q_\gamma}$ after the last join operator, where q_π is the group-by set of the query and q_γ is the set of aggregations functions applied on the features. We remind the reader that the final aggregation or projection is optional.

Example 5. The execution plan of the query in Example 4 is shown in Fig. 4. Noticeably, the aggregation and global join operators directly reference the resolved feature names.

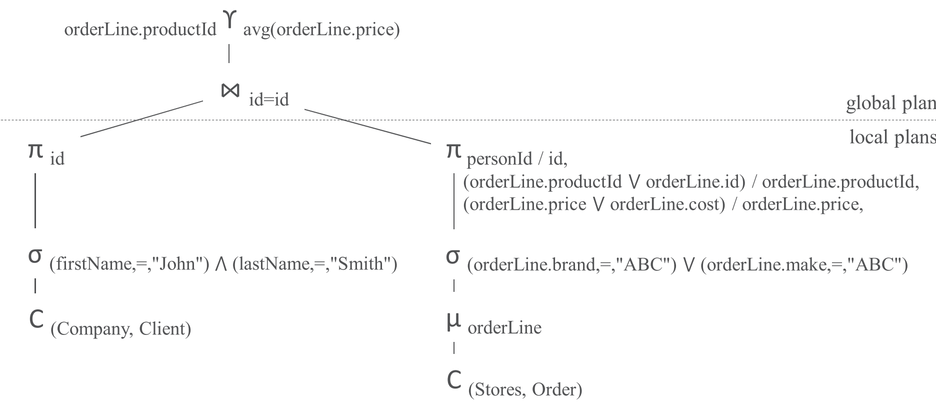

Fig. 5. Execution plan of the query in Example 4.

5 Discussion and Conclusions

Data science and BI 2.0 expect more flexible and lightweight approaches to data analysis. Our proposal extends previous polystore solutions by handling schema heterogeneity and ensuring consistent answer for GPSJ queries, i.e., a wide class of queries that is the most common in OLAP.

Although the main contribution of this paper is the introduction of the formal framework, we carried out a preliminary experimentation through a prototype to verify the correctness and effectiveness of our findings. With reference to Fig. 2 we adopted Spark SQL as the middleware, MySQL, MongoDB and Cassandra as relational, document-based and column-based DBMSs, respectively. The polystore we implemented is based on Unibench and has been extended with schema heterogeneity. All the classes of heterogeneity discussed in the paper have been injected and two different schemata for the Order collection are present. Maximal schema cardinality is 142k records for the Order and Invoice collections. We also defined the minimal set of features to answer a workload of 4 queries. In particular, query in Example 4 (whose plan is reported in Fig. 5) retrieves 23% of the orders, and allows to transparently access the related order lines that are evenly distributed on different schemata of the document DB. Overall query execution requires 6.9 s: 0.2ṡ are necessary to create the plans, 1.2 s to run in parallel the local plans, 2.7 s to generate Spark dataframes and 2.8 s to run the global one. Other queries perform at comparable times and all results correctly correspond to those of a manual execution.

Future extensions will cover different aspects. First, we plan to cover horizontal partitioning of the data, that is, the same collection can span on several collections on potentially different DBs. This introduces a new level of heterogeneity, as features may represent attributes that do not belong to the same collection. We will also extend our approach (1) to support additional data models (e.g., key-value and graph), and (2) to enable a broader set of queries than GPSJs (e.g., [1]). In terms of effectiveness, we will consider the introduction of KPIs to provide further insights to the user with respect to the underlying heterogeneity of the data (e.g., [10]). Finally, we intend to run larger experimentation over real datasets to better study the efficiency and boundaries of our approach.

References

1. Ben Hamadou, H., et al.: Schema-independent querying for heterogeneous collections in NoSQL document stores. Inf. Syst. (2019, in press). https://doi.org/10.1016/j.is.2019.04.005
2. Ben Hamadou, H., Ghozzi, F., Péninou, A., Teste, O.: Towards schema-independent querying on document data stores. In: 20th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data Co-Located with EDBT/ICDT. CEUR-WS.org (2018)
3. Botoeva, E., Calvanese, D., Cogrel, B., Xiao, G.: Expressivity and complexity of MongoDB queries. In: 21st International Conference on Database Theory, pp. 9:1–9:23. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018). https://doi.org/10.4230/LIPIcs.ICDT.2018.9
4. Chang, F., et al.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. **26**(2), 4:1–4:26 (2008)
5. Corbellini, A., Mateos, C., Zunino, A., Godoy, D., Schiaffino, S.N.: Persisting big-data: the NoSQL landscape. Inf. Syst. **63**, 1–23 (2017)

6. DiScala, M., Abadi, D.J.: Automatic generation of normalized relational schemas from nested key-value data. In: 2016 ACM SIGMOD International Conference on Management of Data, pp. 295–310. ACM (2016). https://doi.org/10.1145/2882903.2882924
7. Franklin, M.J., Halevy, A.Y., Maier, D.: From databases to dataspaces: a new abstraction for information management. SIGMOD Rec. **34**(4), 27–33 (2005)
8. Gadepally, V., et al.: The BigDAWG polystore system and architecture. In: 2016 IEEE High Performance Extreme Computing Conference, pp. 1–6. IEEE (2016)
9. Gallinucci, E., Golfarelli, M., Rizzi, S.: Variety-aware OLAP of document-oriented databases. In: 20th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data Co-Located with EDBT/ICDT. CEUR-WS.org (2018)
10. Gallinucci, E., Golfarelli, M., Rizzi, S.: Approximate OLAP of document-oriented databases: a variety-aware approach. Inf. Syst. (2019, in press). https://doi.org/10.1016/j.is.2019.02.004
11. Golfarelli, M., et al.: OLAP query reformulation in peer-to-peer data warehousing. Inf. Syst. **37**(5), 393–411 (2012). https://doi.org/10.1016/j.is.2011.06.003
12. Gupta, A., Harinarayan, V., Quass, D.: Aggregate-query processing in data warehousing environments. In: 21th International Conference on Very Large Data Bases, pp. 358–369. Morgan Kaufmann (1995)
13. Herrero, V., Abelló, A., Romero, O.: NOSQL design for analytical workloads: variability matters. In: 35th International Conference on Conceptual Modeling, pp. 50–64 (2016). https://doi.org/10.1007/978-3-319-46397-1_4
14. Jeffery, S.R., Franklin, M.J., Halevy, A.Y.: Pay-as-you-go user feedback for dataspace systems. In: 2008 ACM SIGMOD International Conference on Management of Data, pp. 847–860. ACM (2008). https://doi.org/10.1145/1376616.1376701
15. LeFevre, J., et al.: MISO: souping up big data query processing with a multistore system. In: 2014 ACM SIGMOD International Conference on Management of Data, pp. 1591–1602. ACM (2014). https://doi.org/10.1145/2588555.2588568
16. Ong, K.W., Papakonstantinou, Y., Vernoux, R.: The SQL++ semi-structured data model and query language: a capabilities survey of SQL-on-Hadoop, NoSQL and NewSQL databases. CoRR abs/1405.3631 (2014)
17. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. VLDB J. **10**(4), 334–350 (2001). https://doi.org/10.1007/s007780100057
18. Rolls, D., Joslin, C., Scholz, S.: Unibench: a tool for automated and collaborative benchmarking. In: 18th IEEE International Conference on Program Comprehension, pp. 50–51. IEEE Computer Society (2010). https://doi.org/10.1109/ICPC.2010.36
19. Rostin, A., et al.: A machine learning approach to foreign key discovery. In: 12th International Workshop on the Web and Databases (2009)
20. Sadalage, P.J., Fowler, M.: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Pearson Education, London (2013)
21. Sheth, A.P.: Federated database systems for managing distributed, heterogeneous, and autonomous databases. In: 17th International Conference on Very Large Data Bases, p. 489. Morgan Kaufmann (1991)
22. Tahara, D., Diamond, T., Abadi, D.J.: Sinew: a SQL system for multi-structured data. In: 2014 ACM SIGMOD International Conference on Management of Data, pp. 815–826. ACM (2014). https://doi.org/10.1145/2588555.2612183
23. Tan, R., et al.: Enabling query processing across heterogeneous data models: a survey. In: 2017 IEEE International Conference on Big Data, pp. 3211–3220. IEEE Computer Society (2017). https://doi.org/10.1109/BigData.2017.8258302

24. Thomas, S.J., Fischer, P.C.: Nested relational structures. Adv. Comput. Res. **3**, 269–307 (1986)
25. Wang, L., et al.: Schema management for document stores. PVLDB **8**(9), 922–933 (2015). https://doi.org/10.14778/2777598.2777601
26. Zaharia, M., et al.: Apache Spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016). https://doi.org/10.1145/2934664