The VLDB Journal (2021) 30:1017–1040
https://doi.org/10.1007/s00778-021-00682-5

REGULAR PAPER

A dataspace-based framework for OLAP analyses in a high-variety multistore

Chiara Forresi[1] · Enrico Gallinucci[1] · Matteo Golfarelli[1] · Hamdi Ben Hamadou[2]

Received: 31 July 2020 / Revised: 23 April 2021 / Accepted: 19 May 2021 / Published online: 31 July 2021

Abstract
The success of NoSQL DBMSs has pushed the adoption of polyglot storage systems that take advantage of the best characteristics of different technologies and data models. While operational applications take great benefit from this choice, analytical applications suffer the absence of schema consistency, not only between different DBMSs but within a single NoSQL system as well. In this context, the discipline of data science is steering analysts away from traditional data warehousing and toward 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 high-variety multistore, with heterogeneous schemas and overlapping records. Our approach supports relational, document, wide-column, and key-value data models by automatically handling both data model and schema heterogeneity through a dataspace layer on top of the underlying DBMSs. 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 system has been prototyped on Apache Spark.

Keywords Multistore · 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 toward *polyglot persistence* [1] (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 toward 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 [2], where the integration is 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 terms *data virtualization* and *data fabric*[1] have born to identify solutions that transparently access multiple and heterogeneous sources to accelerate digital transformation and to support multi-cloud architectures. Commercial software, such as Denodo [3], implements data virtualization in many different contexts and supports both operational and analytical applications.

The main challenges to address in this context are related to (i) the heterogeneity of the data in terms of data model and schema and (ii) the overlap of the same data across differ-

✉ Enrico Gallinucci
enrico.gallinucci@unibo.it

Chiara Forresi
chiara.forresi@unibo.it

Matteo Golfarelli
matteo.golfarelli@unibo.it

Hamdi Ben Hamadou
hamdibh@cs.aau.dk

1 University of Bologna, Cesena, Italy

2 Aalborg University, Aalborg, Denmark

[1] Data fabric is an architecture and set of data services that provide consistent capabilities across a choice of endpoints, spanning from on-premises to multiple cloud environments.

ent collections of data (i.e., record overlapping). *Data model heterogeneity* is intrinsic in a polyglot database; it requires distributing the computation of a query across the different databases (which adopt different query languages) and possibly relying on 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 RDBMSs (which requires all record in a table to comply with a predefined schema) in favor of a *soft-schema* approach, in which each record embeds its own schema definition. The most typical schema variants consist of missing or additional attributes, different naming conventions or data types for an attribute, and different record structures. Schema heterogeneity is mainly due to schema evolution and to the acquisition of data from sources adopting different schema representations for the same entities. Additionally, due to *record overlapping* the same record may exist in different collections; this problem occurs when two distinct collections (possibly in different databases) model the same real-world entity (e.g., a set of customers) in a non-partitioned way (e.g., the collection modeling customers from grocery stores may overlap the collection modeling customers from the e-commerce application). The implication is that records belonging to the same entity must be reconciled (or merged) to avoid data replication and solve potential conflicts (e.g., the customer's name on one collection may differ from its name on the other one). An exemplification of these problems is given in Fig. 1, where overlapping records of customers and orders from two DBMSs (relational and document-based) need to be reconciled in order to obtain a clean representation that can be used for analyses purposes. Notice the overlap of customer 123 and order O1 in different schema representations; orders have different attributes, customers have different naming conventions and conflicting values for name and age.

State-of-the-art proposals for polyglot systems mainly include *multistores* and *polystores*, depending on whether they provide single or multiple interfaces for cross-DBMS querying [4]. Current solutions mostly focus on addressing data model heterogeneity and on optimizing the query processing, but they do not consider schema heterogeneity nor record overlapping. This prevents analysts from taking full advantage of the data, as several instances may be missed by queries that do not consider schema variations, and query results may be inconsistent.

The proposal of this paper is an extension of our recent research effort in this direction [5] to define an approach that supports data analysis within a multistore by handling data model heterogeneity, schema heterogeneity, and record overlapping 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 [6]. 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 [7]), 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 query semantics to prevent the user from getting misleading results leading to ambiguous or potentially incorrect interpretation in the analytical context. For a given GPSJ, our approach defines a cross-database execution plan in nested relational algebra (NRA) [8], which is compatible with the expressiveness of document stores' query language [9] and SQL (as it is a superset of relational algebra), with the latter being used by both RDBMSs and wide-column systems. The execution plan handles both record overlapping and schema heterogeneity by, respectively, relying on the *merge* operator (i.e., a new NRA operator that we introduce to enable conflict-resolution) and by relying on the dataspace knowledge in terms of mappings between the collections' attributes. A prototype of the approach has been implemented in Scala with Apache Spark, i.e., a Big Data framework that enables the execution of collection plans on the single databases (i.e., PostgreSQL, MongoDB, Cassandra, and Redis in our prototype), the in-memory elaboration of intermediate results, and the capability to scale to large amounts of data. We remark that the approach does not modify the original data, thus ensuring the validity of existing workloads on the databases while granting access to the dataspace. Finally, an experimental evaluation of the approach is carried out to measure it both in terms of efficiency and effectiveness.

The main original contributions of this paper can be summarized as follows.

- We propose an approach that relies on a dataspace to support data analysis within a multistore by handling not just data model heterogeneity and schema heterogeneity but also record overlapping. To this end, we introduce a new NRA operator to handle conflict-resolution between overlapping records and we significantly revise and extend both the formalization and the algorithmic logic initially proposed in [10].
- We formalize and discuss the algorithms to produce an execution plan from a GPSJ query formulated on the dataspace, including a set of applied optimizations.
- We present a prototypical implementation of the approach on which we carry out an extensive experimental evaluation in both terms of efficiency and effectiveness.

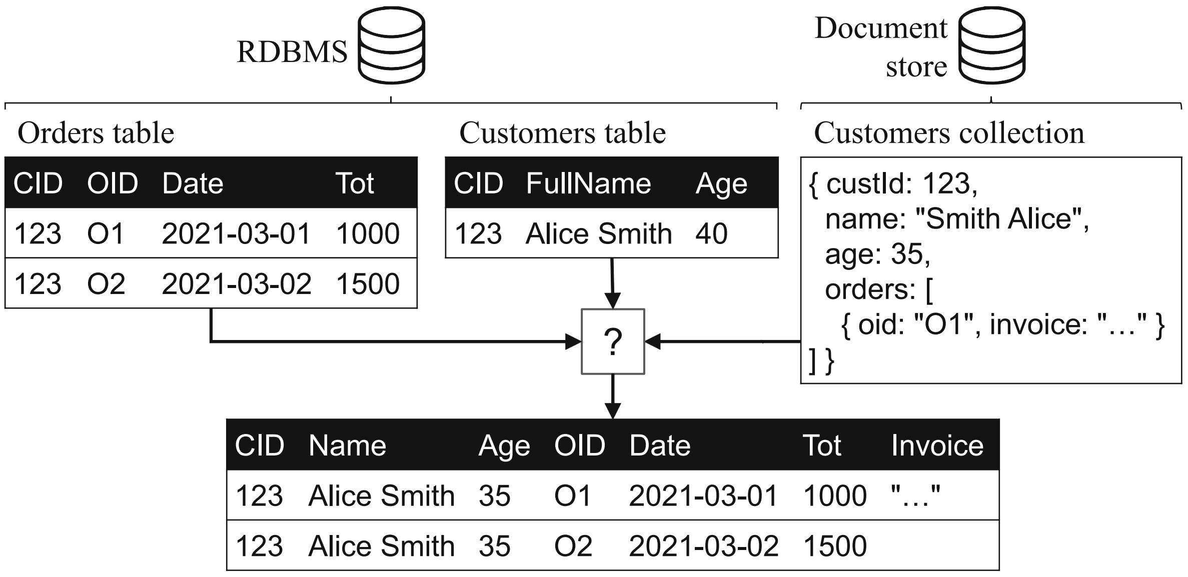


Fig. 1 An exemplification of data model heterogeneity, schema heterogeneity, and record overlapping in a multistore

The paper outline is as follows. Section 2 presents the use cases of our approach and discusses the case study. The dataspace and the basic concepts are formalized in Sect. 3. Then, we present the formulation of the execution plan in Sect. 4 and discuss the experimental evaluation in Sect. 5. After discussing the related work in Sect. 6, we draw the conclusions and discuss future work in Sect. 7.

2 Use cases & case study

The approach proposed in this work can be applied in different practical contexts which, as mentioned in the introduction, refer to data virtualization systems for data analysis. Below we describe two more specific contexts that emerged during our interaction with Denodo [3], one of the market-leading tools on this subject.

– *Analytical data offloading*: to reduce costs and optimize performance, the historical depth of databases is kept limited; typically, it is 1-2 years for operational systems, and 3-5 for analytical ones [11]. After these periods, data are offloaded to cheaper as well as bigger storages, such as cloud storages or data lakes. Offloading implies a change of data model, a change of schema, and obviously an overlapping of instances with the original data. For example, offloading a relational data warehouse could imply turning instances stored in a star schema to a single JSON document including both measures and dimensional attributes; alternatively, a relational flat schema could be adopted. Similarly, invoices stored in an ERP can be offloaded to a key-value repository, where the value stores an object including only the attributes relevant for fiscal purposes. In the meanwhile, the in-place data may evolve in terms of structures or values. In this context, unforeseen analyses are often needed, such as data enthusiasts asking to compare the offloaded data with the in-place ones.
– *Multi-cloud architecture*: this context combines different storage technologies and resources from multiple cloud platforms [12]. It allows application providers to manage the risks associated with technology, vendor lock-in, provider reliability, data security, and privacy thus, it is an increasingly popular tactic for designing the storage tier of cloud-based applications [13]. The multi-cloud architecture and related frameworks (e.g., data fabric) accelerate digital transformation since they enable the exploitation of data spread across different providers and architectures, all the while overcoming data silos through data virtualization. Multi-cloud architectures are a *panacea* in presence of many company branches. For example, consider a holding or a federation of companies (e.g., hospitals in the health sector). In this case, a lot of data is shared between the branches, but each branch is free to choose its own storage provider (either on cloud or on-premise), data model, and schema. To keep it simple, let us consider the case of ICD-9-CM (International Classification of Diseases) [14], which is often used in OLAP analysis in the healthcare domain. ICD-9-CM changes some of its attributes and values across the years; thus, depending on the ICD-9-CM version adopted by each branch, data overlapping and schema heterogeneity must be resolved when cross-queries are issued over the branches' databases. Furthermore, every hospital or local

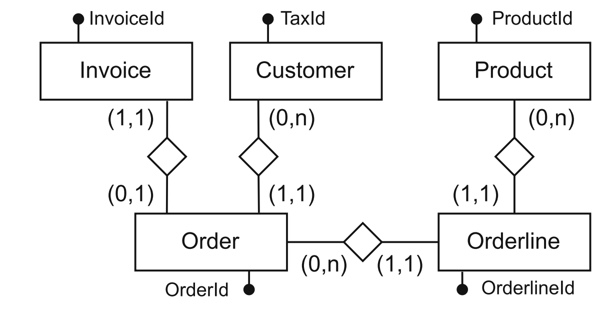


Fig. 2 The ER diagram of the case study

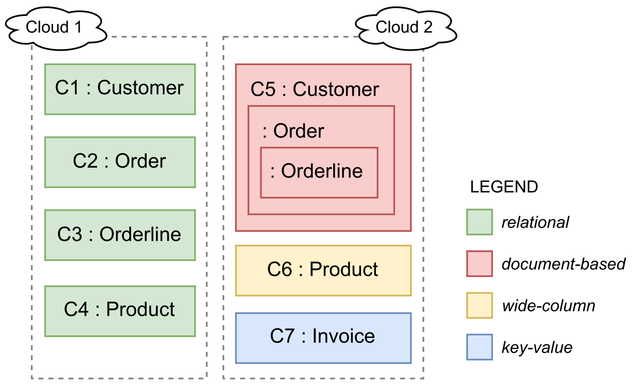


Fig. 3 A graphical representation of the physical implementation of the case study. Different colors represent different DBMSs with different data models

health unit can store such data in different data models and schemas, depending on the adopted software.

The use cases above are characterized by (1) multiple data models, (2) heterogeneous schemas, and (3) overlapping instances. This proves the relevance of the discussed issue, which will further increase with the progressive diffusion of data virtualization architectures.

To analyze the three characterizing features we rely on a variation of Unibench [15], i.e., a benchmark dataset for multi-model databases based on an e-commerce application, which stores details about products ordered and bought by customers. A conceptual view of the use case is shown in Fig. 2 through an ER diagram. The case study is perfectly suited for GPSJ queries since it models events (e.g., Orderline and Order), KPIs or measures (e.g., quantity and Price), and grouping/classification attributes (e.g., Product and Customer).

The case study simulates the multi-cloud architecture, where different branches of the same holding basically store the same data but rely on different storage systems. Figure 3 shows the physical implementation. C_1 to C_7 represent the collections of data, while the ":" notation is used to indicate the granularity of the data in the collection (notice that the document-based database contains a single collection which uses nested structures to embed orders within customers, and order lines within orders). While Cloud 1 fully relies on a relational DBMS, Cloud 2 satisfies a need of supporting data variety by relying on NoSQL systems; also notice that Cloud 2 additionally stores orders' invoices. With respect to the aforementioned use cases, our case study is characterized by the same features.

– *Multiple data models*: being a multistore, it comprises databases in four data models: relational, document-based, key-value, and wide-column.
– *Heterogeneous schemas*: given the schemaless nature of NoSQL databases, the collections in Cloud 2 are characterized by varying levels of schema heterogeneity; in particular, we have 35 schemas in C_5 and 2 schemas C_6, while invoices in C_7 are free to be in any schema.
– *Overlapping instances*: as the two branches belong to the same holding, both customers and products are partially overlapped in the two cloud environments.

To fulfill these characteristics, the Unibench benchmark is extended by injecting schema heterogeneity and introducing overlapping records in different DBMSs. In particular, we carry out the following extensions.

– Unibench's customer records are split between C_1 and C_5 in an overlapping fashion: 90% customers are in the relational table and 30% in the document collection, meaning that 20% of the customers are replicated.
– Unibench's product records are split between C_4 and C_6 in an overlapping fashion: in this case, both the relation table and the wide-column collection contain 80% of the original records, where 60% of the products are replicated.
– Unibench's order and order line records are split between the two DBMSs, but they do not overlap (i.e., each record exists in one copy only). Orders belonging to overlapping customers are randomly assigned to one of the branches.
– Missing attributes are introduced in C_5 and C_6; in the former, we remove values for three attributes (namely browser, locationIP, and place) for some random customer records, in the latter we remove imgUrl values in 10% of the product records.
– Semantic equivalence is introduced in C_5 by renaming the birthdate and gender attributes into dateOfBirth and sex for some random customer records, and by renaming all attributes into a different convention for some random order line records.
– Different data types are introduced in C_1 and C_5 by storing the value of attribute OrderDate as a date in the former's records and as a string in the latter's; also, order lines' attribute Quantity in C_5 is randomly modeled as a string or a number.

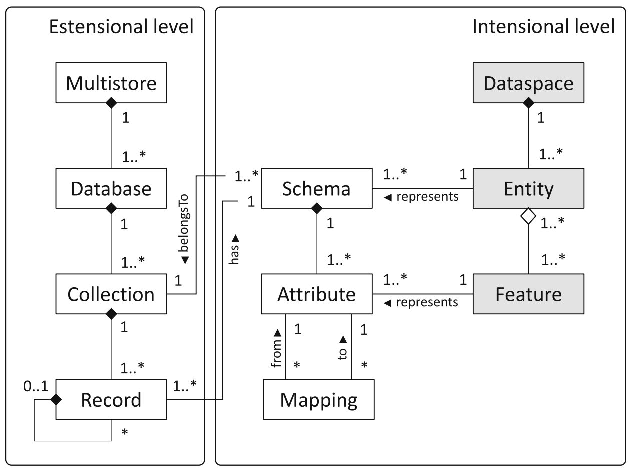


Fig. 4 A UML class diagram of the terminology

These extensions define a multistore with fixed levels of schema heterogeneity and record overlapping. Different implementations of the multistore will be defined in Sect. 5 to evaluate these specific aspects.

3 The dataspace

This section is focused on the formalization and presentation of the dataspace. First, in Sect. 3.1 we introduce the intensional representation of existing collections; then, we formalize the dataspace in Sect. 3.2; finally, Sect. 3.3 describes the process to obtain, maintain, and use the dataspace.

3.1 Modeling the existing collections

In a multistore, different data models may be used to represent and store data. This requires defining the basic database concepts in a way that abstracts from the single data models. Figure 4 provides a UML class diagram to describe the concepts that we use in this paper.

Definition 3.1 (Multistore, Database, Collection) A multistore is a set of databases; each database D is a set of collections; each collection C is a set of records.

We use the term *collection* to refer to the container of data (i.e., what is known as *table*, *column family*, and *collection* in relational, wide-column, and document/key-value databases, respectively). Similarly, we introduce the term *record* to refer to the instances in a collection. Our notion of records perfectly corresponds to the *tuples* of a relational database, but the *rows* and *documents* of wide-column and document databases potentially correspond to multiple records. In fact, non-relational data models comply with the *aggregate data modeling* property, which enables the *nesting* of records within other records through the array data type. Thus, we do not consider documents and rows as a whole, but we separately model the records available at each nesting level.

Definition 3.2 (Record, Attribute) A record $r = \{v_1, \dots, v_n\}$ is a set of values, and each value v_i is associated with a certain attribute a_i. Let $r[a_i] = v_i$ where a_i is an attribute; v_i is either a value of primitive type (e.g., number or string) or an array of records. An attribute a is defined by a name and a type (i.e., either primitive or array).

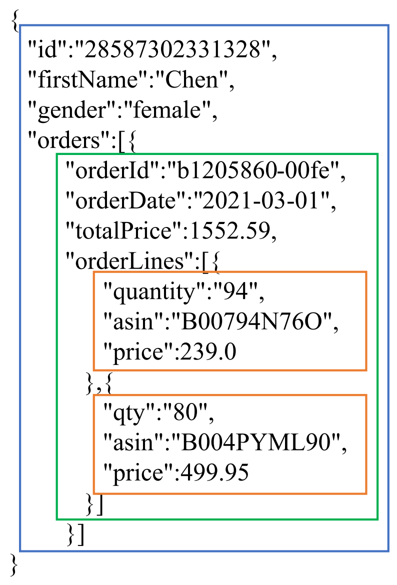
From this point forward, we refer to *primitive attributes* or *array attributes* based on the respective type. Also, given an attribute a, we use $name(a)$ and $type(a)$ to, respectively, refer to its name and its type. If an attribute is nested within one or more array attributes, its name includes the dot-concatenation of the names of those array attributes. Finally, notice that: (i) arrays of primitive types are not considered for simplicity; (ii) attributes of object data type are not considered as they are simply containers of attributes for the same record (i.e., they entail the same expressiveness of primitive attributes); (iii) we exclude attributes of binary data types, whose values are uninterpretable without additional knowledge; (iv) to enable support to key-values stores (which support only two attributes, i.e., a string key and a binary value), we assume the value to contain interpretable strings; in particular, the name of the two attributes are inferred from the collection's name (e.g., collection C_7 in Fig. 3 has two attributes a_k and a_v, where $name(a_k) =$ invoiceId, $name(a_v) =$ invoice, and $type(a_k) = type(a_v) =$ string).

Example 3.1 Figure 5 shows a sample document of a document database, representing a customer, its orders, and the respective order details; the boxes highlight the presence of four records (in blue the customer record, in green the order records, in orange the two order line records).

Our notion of *schema* applies to the records rather than to the entire collections. Thus, several schemas may be found for a certain collection, due to the possible presence of both schema variability and nested records.

Definition 3.3 (Schema, Key) A schema S applies to one or more records in a collection and it is defined as a set of primitive attributes. The attribute that uniquely identifies the records with schema S is the *key*, defined as $key(S)$. If the records referring to S are nested, S^μ denotes the optional sequence of array attributes in the schema's collection that must be unnested to unveil the records of S.

For the sake of simplicity, given a record r, its schema (denoted with S_r) is the set of attributes directly available in r (i.e., without unnesting any array). If a record r' is nested within r, $S_{r'}$ also includes $key(S_r)$; this is necessary to maintain the relationship between the schema of a nested record



```
{
  "id":"28587302331328",
  "firstName":"Chen",
  "gender":"female",
  "orders":[{
    "orderId":"b1205860-00fe",
    "orderDate":"2021-03-01",
    "totalPrice":1552.59,
    "orderLines":[{
      "quantity":"94",
      "asin":"B00794N76O",
      "price":239.0
    },{
      "qty":"80",
      "asin":"B004PYML90",
      "price":499.95
    }]
  }]
}
```

Fig. 5 A sample document representing a customer, its orders, and the respective order details; four *records* are shown in the boxes, and each color (blue, green, and orange) corresponds to a different *schema*

and the one of the parent record. Our schema definition provides a view of the records in first normal form, as it hides the denormalization due to the nesting of records and exposes the relationships between schemas at different nesting levels.

For the sake of simplicity, we assume all keys to be simple (i.e., not composite)[2]. Also, it is reasonable to assume that all schemas (including those nested in arrays) have a key. We refer to $\mathcal{S}$ as the set of all schemas within the multistore.

Example 3.2 The sample document in Figure 5 contains three schemas:

- S_{blue} = {id, firstName, gender}
- S_{green} = {id, orders.orderId, orders.orderDate, orders.totalprice}
- S_{orange} = {orders.orderId, orders.orderLines.quantity, orders.orderLines.asin, orders.orderLines.price}

It is S^{μ}_{blue} = [], S^{μ}_{green} = [orders], and S^{μ}_{orange} = [orders, orders.orderLines]

Concerning the schemaless property of non-relational databases, we take into account every schema variation in a collection (i.e., if two records differ even for a single attribute, we model two separate schemas, each with its own set of attributes). Given our previous assumptions, collections in key-value stores are associated with a single schema (e.g., the schema of collection C_7 is S = {invoiceId, invoice}).

[2] Composite keys could be supported by extending the definitions of schemas, keys, and mappings; however, assume simple keys to ensure better readability of the paper.

3.2 Modeling the dataspace

Due to both schema variability and schema denormalization, attributes in different schemas may represent the same property. For example, in Fig. 5 different order line records use attributes with different names to indicate the quantity of product bought (i.e., quantity and qty, respectively). To resolve the different classes of heterogeneity and model the equivalence between different attributes of the dataspace we exploit mappings[3].

Definition 3.4 (Mapping) A mapping m is a triple $m = (a_i, a_j, \varphi_{(a_i,a_j)})$ that expresses a relationship between two primitive attributes a_i and a_j; $\varphi_{(a_i,a_j)}$ is an bijective transcoding function to express the values of a_j in the format of a_i (if necessary; otherwise, $\varphi_{(a_i,a_j)} = I()$ where $I()$ is the identity function). The existence of a mapping between a_i and a_j is indicated with $a_i \equiv a_j$.

For simplicity, we consider only simple mappings between two attributes. Since mappings are specified between attributes of different schemas, they reveal the relationship between such schemas. Consider two schemas S_i and S_j.

- If $key(S_i) \equiv key(S_j)$, then we infer a *one-to-one* relationship, represented as $S_i \leftrightarrow S_j$.
- If $a_k \equiv key(S_j) : a_k \in \{S_i \setminus key(S_i)\}$, then we infer a *many-to-one* relationship from S_i to S_j, represented as $S_i \xrightarrow{a_k} S_j$.
- If $a_k \equiv a_l : (a_k, a_l) \in (\{S_i \setminus key(S_i)\}, \{S_j \setminus key(S_j)\})$, no direct relationship exists between the two schemas.

Mappings recognize that there is a semantic equivalence between two attributes in different schemas, thus we need to address all of them through a unique reference. This is the purpose of features.

Definition 3.5 (Feature) A feature represents either a single attribute or a group of attributes that are mapped to each other. We define a feature as $f = (a, name, M, \mathbb{\Lambda})$, where a is the *representative* attribute of the feature; $name$ is the name of the feature (possibly different from $name(a)$); M is the set of mappings that link all the feature's attributes to the representative a (i.e., the transcoding functions in the mapping are all directed toward a); $\mathbb{\Lambda} : (v_i, v_j) \rightarrow v_k$ is an associative and commutative function that resolves the possible conflict between the values of any two attributes (a_i, a_j) belonging to f and returns a single value v_k. Function $\mathbb{\Lambda}$ may either choose one value between v_i and v_j or calculate a new value v_k. It is $M = \varnothing$ when a concept is modeled by a single attribute.

[3] Unlike in [10], the mapping definition in this paper has not been specialized into *sameAs* and *fk* since they are not necessary to the query rewriting process.

Let $attr(f)$ be the set of attributes represented by f (i.e., the representative attribute plus those derived from the mappings). Given a record r, the conflict-resolution function $\bigwedge\!\!\!\bigwedge$ can be applied to $r[a_i]$ and $r[a_j]$ if $\{a_i, a_j\} \subseteq attr(f)$; we refer the reader to [16] for an indication about different methods to define conflict-resolution functions. Also, we remark that an attribute is always represented by one and only one feature; thus, for any two features f_i and f_j, it is $attr(f_i) \cap attr(f_j) = \varnothing$. We use $feat(a)$ to refer to the feature of an attribute a, $name(f)$ to refer to the name of a feature, $rep(f)$ to refer to the representative attribute of f, and $rep(a)$ as short for $rep(feat(a))$.

Similarly to attributes, several schemas may be found to represent the same semantic concept (e.g., customers, orders). To hide such structural complexity, we introduce the concept of entities.

Definition 3.6 (Entity) An entity is a representation of a set of schemas in the multistore that semantically model the same semantic concept. We define an entity as $E = (name, \mathcal{S}_E, \phi_E)$, where $\mathcal{S}_E \subseteq \mathcal{S}$ is the set of schemas represented by E, and ϕ_E is a Boolean variable that indicates whether the schemas in $\mathcal{S}_E$ are subject to record overlapping. The schemas in $\mathcal{S}_E$ must be in a one-to-one relationship with each other, i.e., $\forall\ (S_i, S_j) \in \mathcal{S}_E$, it is $key(S_i) \equiv key(S_j)$, i.e., $\exists f : attr(f) \supseteq \{key(S) : S \in \mathcal{S}_E\}$

Example 3.3 While Fig. 3 presents the collections and schemas in our motivating example, Table 1 presents a detailed view in terms of schemas, attributes, features and entities. On the columns, the schemas are organized by entities; on the rows, attributes are organized by features, and the mappings are implicit between attributes of the same feature. For instance, it is $a_7 \equiv a_8$ since $feat(a_7) = feat(a_8) = f_2$. Mappings reveal the relationship between the schemas. For instance, $S_1 \leftrightarrow S_2 \leftrightarrow S_{10}$ because $key(S_1) \equiv key(S_2) \equiv key(S_{10})$; similarly, it is $S_5 \leftrightarrow S_6 \leftrightarrow S_9$. Differently, mapping $a_3 \equiv a_4$ indicates that $S_5 \xrightarrow{a_4} S_1$ because $a_3 \neq key(S_5)$ and $a_4 = key(S_1)$. We remark that S_1 differs from S_2 on the existence of an attribute representing f_3, and that S_4 differs from S_8 on the datatype of f_9 (although this in not made explicit in Table 1 for space reasons, a_{29} is modeled as a string, while a_{30} as a date). Ultimately, notice that (i) each attribute is contained only in one schema, (ii) each schema contains one key attribute, (iii) each schema contains at most one attribute per feature, and (iv) there exist several features (e.g., ProductId) that overlap several entities.

Ultimately, the dataspace is the data structure that puts together features and entities.

Definition 3.7 (Dataspace) A dataspace $\mathcal{D}$ is a graph of entities and features, where each feature is connected to the entities whose schemas contain an attribute of such feature.

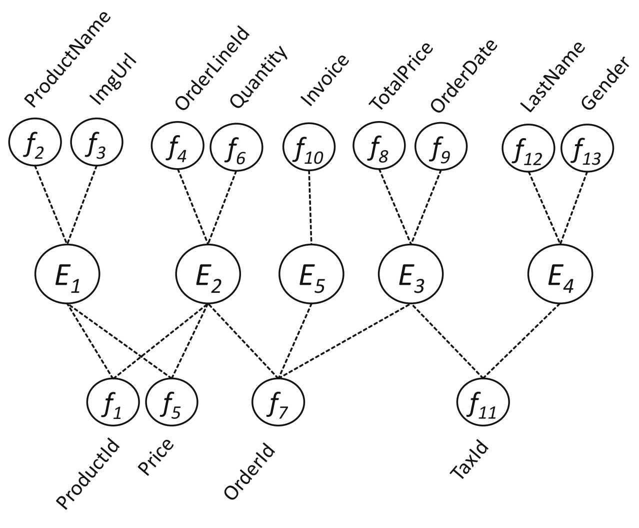


Fig. 6 The dataspace $\mathcal{D}$ of the running example

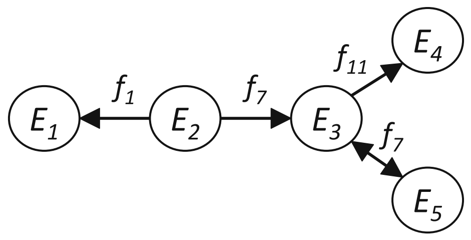

$$\begin{tikzcd}
 & & & E_4 \\
E_1 & E_2 \arrow[l, "f_1"'] \arrow[r, "f_7"] & E_3 \arrow[ur, "f_{11}"] \arrow[dr, leftrightarrow, "f_7"] & \\
 & & & E_5
\end{tikzcd}$$

Fig. 7 The entity graph $G^{\mathcal{E}}$ of the running example

To emphasize the relationships between the entities and exploit them for querying purposes, we organize them in a supporting structure called *entity graph*.

Definition 3.8 (Entity graph) The entity graph is a directed acyclic graph $G^{\mathcal{E}} = (\mathcal{E}, L^{\mathcal{E}})$ where $\mathcal{E}$ is the set of entities in the dataspace and $L^{\mathcal{E}}$ is the set of -to-one relationships (or links) between the entities.

We say that $E_i \xrightarrow{f} E_j$ if $\exists\ f \in E_i : \forall\ a \in attr(f), a \in S_{E_i}$ it is $S_{E_i} \xrightarrow{a} S_{E_j}, (S_{E_i}, S_{E_j}) \in (\mathcal{S}_{E_i}, \mathcal{S}_{E_j})$. In other words, there is a many-to-one relationship from E_i to E_j on f if $\forall\ S_{E_j} \in \mathcal{S}_{E_j}$ it is $attr(f) \cap key(S_{E_j}) \neq \varnothing$ (i.e., the attributes of f are keys in the schemas of E_j) and $\forall\ S_{E_i} \in \mathcal{S}_{E_i}$ it is $attr(f) \cap key(S_{E_i}) = \varnothing$ (i.e., the attributes of f are not keys in the schemas of E_i). Similarly, we say that $E_i \overset{f}{\longleftrightarrow} E_j$ if $\exists\ f \in E_i : \forall\ a \in attr(f), a \in S_{E_i}$ it is $S_{E_i} \overset{a}{\longleftrightarrow} S_{E_j}, (S_{E_i}, S_{E_j}) \in (\mathcal{S}_{E_i}, \mathcal{S}_{E_j})$. Notice that we do not consider many-to-many relationships because schemas are inferred from physical implementations, where only many-to-one can be explicitly represented.

Example 3.4 Figures 6 and 7, respectively, show the dataspace and the entity graph of the running example.

Table 1 Extract of the correspondences between attributes and schemas in our case study from Fig. 3; cell $[i, j]$ has a checkmark if $a_i \in S_j$, or the letter “K” if $a_i = key(S_j)$. Attributes are organized by features f_k and indicate the collection C_l they belong to, while schemas are organized by entity E_m

name(f)	f	a	C	Product E_1			Orderline E_2			Order E_3		Customer E_4		Inv E_5
				S_1	S_2	S_{10}	S_5	S_6	S_9	S_4	S_8	S_3	S_7	S_{11}
ProductId	f_1	a_1	C_3						✓					
		a_2	C_5					✓						
		a_3	C_5				✓							
		a_4	C_6	K										
		a_5	C_6		K									
		a_6	C_4			K								
ProductName	f_2	a_7	C_6	✓										
		a_8	C_6		✓									
		a_9	C_4			✓								
ImgUrl	f_3	a_{10}	C_6	✓										
OrderLineId	f_4	a_{11}	C_3						K					
		a_{12}	C_5				K							
		a_{13}	C_5					K						
Price	f_5	a_{14}	C_5				✓							
		a_{15}	C_5					✓						
		a_{16}	C_6	✓										
		a_{17}	C_6		✓									
Quantity	f_6	a_{18}	C_3						✓					
		a_{19}	C_5				✓							
		a_{20}	C_5					✓						
OrderId	f_7	a_{21}	C_3						✓					
		a_{22}	C_3								K			
		a_{23}	C_5				✓							
		a_{24}	C_5					✓						
		a_{25}	C_5							K				
		a_{26}	C_7											K
TotalPrice	f_8	a_{27}	C_2								✓			
		a_{28}	C_5							✓				
OrderDate	f_9	a_{29}	C_2								✓			
		a_{30}	C_5							✓				
Invoice	f_{10}	a_{31}	C_7											✓
TaxId	f_{11}	a_{32}	C_1										K	
		a_{33}	C_2								✓			
		a_{34}	C_5									K		
		a_{35}	C_5							✓				
LastName	f_{12}	a_{36}	C_1										✓	
		a_{37}	C_5									✓		
Gender	f_{13}	a_{38}	C_1										✓	
		a_{39}	C_5									✓		

3.3 Obtaining the dataspace

The iterative process to obtain, maintain, and use the dataspace is described in Fig. 8; the figure distinguishes the *offline* activities related to the management of the dataspace (in gray) from the *online* querying activity that relies on the dataspace (in white). Each step is described in the following.

Schema extraction. This step is aimed at extracting schemas from the collections in each database and retrieving collections’ statistics. Its execution is completely automatic

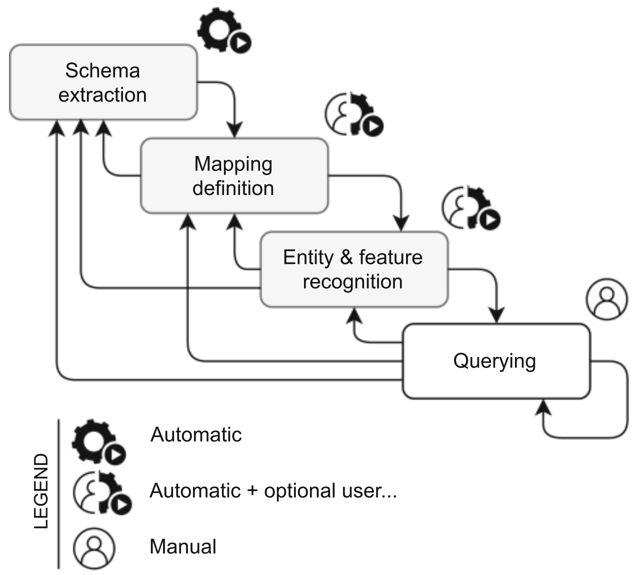


Fig. 8 The process to obtain, maintain, and use the dataspace

and can be carried out incrementally, i.e., new/updated collections can be examined individually at any time.

Mapping definition. The goal of this step is to define mappings between the extracted schemas and attributes. This can be achieved in a semiautomatic manner, i.e., by combining the results of a schema matching algorithm [17] or tool (e.g., Coma 3.0 [18]) with knowledge manually provided by the user. In accordance with the pay-as-you-go philosophy, this methodology enables users to quickly reach the querying step; the mappings automatically defined by the algorithms are later refined by the user, as new insights are obtained through the querying of data.

Feature and entity recognition. This step is semiautomatic as well: based on Definitions 3.5 and 3.6 , both features and entities are automatically derivable from the mappings and the one-to-one relationships between schemas, respectively. Then, the user may refine the results by verifying whether or not the structural one-to-one relationships between schemas actually correspond to the same semantic concept. For instance, in our case study, Order schemas are in a one-to-one relationship with Invoice schemas, but they correspond to different semantic concepts. Similarly, given an entity E, ϕ_E can be set manually or by running an automatic procedure that looks for matches between the key values across $\mathcal{S}_E$.

Querying. As soon as the dataspace is built, the user can exploit it to query the data; details on the querying step are given in Sect. 4. At any point in time, the user can go back to any of the previous steps to re-run some algorithm or to inject knowledge into the system.

The level of user intervention required in the semiautomatic activities of the offline phase is expected to decrease as users advance in the dataspace definition process: a core part of the dataspace will be stabilized after some iterations and it will be updated based on new user requirements (e.g., the exploration of attributes or schemas that had not been analyzed before), schema evolution, or the addition of new data sources. The cost of the update process remains constant over time, because (i) it only consists of updates at the metadata level, and (ii) it is safe to assume that existing constructs of the dataspace will not need to be redefined (if not to make corrections).

4 Execution plan formulation

This section describes the core aspect of our approach, i.e., the formulation of a query by the user on the dataspace and the rewriting process to execute it. In this work, we consider the class of GPSJ queries, formulated on the features available in the dataspace.

Definition 4.1 (Query) Let F be the set of features in a dataspace $\mathcal{D}$; we define a query as $q = (q_\pi, q_\gamma, q_\sigma)$, where: $q_\pi \subseteq F$ specifies the optional set of features to be projected; q_γ specifies optional aggregations as a set of couples (f, op), where $f \in F$ and op is an aggregation function (e.g., $max()$); q_σ is an optional set of conjunctive ($\wedge$) selection predicates in the form of triplets (f, ω, v), where $f \in F$, $\omega \in \{=; >; <; \neq; \geq; \leq\}$ and v is a value[4]. Clearly, at least one among q_π and q_γ must be defined.

GPSJ expressions extend select-join expressions with aggregation, grouping, and group selection. GPSJ queries are the most common class of queries in OLAP applications. It is not mandatory that all the three sets q_π, q_γ and q_σ are present, thus our definition also covers simple selection queries and join queries.

Example 4.1 Let q be the query to measure, for each ProductName, the average Quantity bought by female customers (Gender) starting from 2019 (OrderDate). The group-by set of q is $q_\pi = \{f_2\}$; the aggregation set is $q_\gamma = \{(f_6, avg())\}$ and the set of selection predicates is $q_\sigma = \{(f_9, \geq, \text{“}2019/01/01\text{''}), (f_{13}, =, \text{“}F\text{''})\}$.

The remainder of the section is organized as follows: Sect. 4.1 introduces our extended version of nested relational algebra; Sect. 4.2 describes the query rewriting process; Sect. 4.3 discusses the query plan optimization aspect.

4.1 NRA and the merge operator

We rely on nested relational algebra (NRA) to define the execution plan of a query. Table 2 briefly explains each operator. With respect to traditional algebra, we introduce a new

[4] Disjunctive selection predicates and negations of selection predicates are not supported to avoid overcomplicating the discussion.

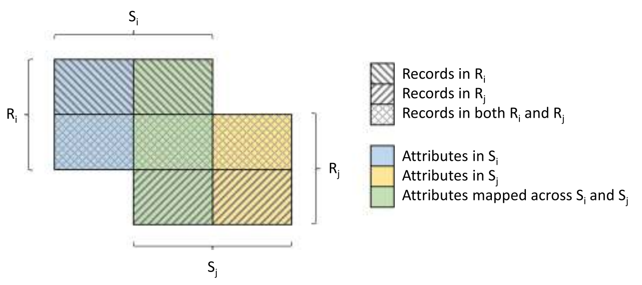


Fig. 9 Graphical representation of the merge operator

operator called *merge* ($\sqcup$), i.e., an adaptation to our scenario of the *full outerjoin-merge* operator introduced in [19]. Its purpose is to replace the join operator ($\bowtie$) by addressing the extensional and intensional overlap between schemas. In particular, we consider the scenario in which records belonging to the same entity (e.g., Customer) can be partially overlapped, both in terms of instances (e.g., the same customer can be repeated across different schemas) and in terms of schemas (e.g., the name of the customer can be an attribute of two different schemas). We assume that the same is true when joining records from different entities (e.g., Customer and Order): the records can be partially overlapped (e.g., a customer may not have any orders, and an order may not be related to any customer), and so can be their schemas (e.g., the name of the customer can be used in the order's schema as well).

Example 4.2 This scenario is explained in Fig. 9, where two overlapping schemas S_i and S_j with the, respectively, overlapping sets of records R_i and R_j are shown. The (vertical) green section is the intersection of schemas, i.e., $S_i \cap S_j = \{(a_k, a_l) \in (S_i, S_j) : a_k \equiv a_l\}$. The (horizontal) crossed section is the intersection of records, i.e., $R_i \cap R_j = \{r : \exists\,(s,t) \in (R_i, R_j), s[a_k] = t[a_l]\}$ where $a_k = a_l$ is the join condition between records s and t.

We aim to keep as much information as possible when joining the records of two schemas, both from the extensional and the intensional points of view. The merge operator ($\sqcup$) answers this need by (i) avoiding any loss of records, (ii) resolving mappings by providing output in terms of features instead of attributes, and (iii) resolving conflicts whenever necessary.

Definition 4.2 (Merge operator) Let R_i and R_j be the recordsets of two schemas S_i and S_j, and consider $(a_k, a_l) \in (S_i, S_j)$ such that $a_k \equiv a_l$, i.e., $\exists f : \{a_k, a_l\} \subseteq attr(f)$. The merge of the two schemas $S_i \sqcup_f S_j$ produces a recordset R_{ij} with schema $S_{ij} = S_i^* \cup S_j^* \cup S_{ij}^{\cap}$ such that:

- $S_i^* = \{a \in S_i : \nexists\, a' \in S_j, a \equiv a'\}$
- $S_j^* = \{a' \in S_j : \nexists\, a \in S_i, a \equiv a'\}$
- $S_{ij}^{\cap} = \{rep(a)\ \forall\,(a, a') \in (S_i, S_j) : a \equiv a'\}$

Table 2 NRA operators

Operator	Description
CA_{col}	Denotes the access to the records of collection *col*.
$\mu_a(C)$	Denotes the unnesting of an array attribute a on collection C.
$\sigma_x(C)$	Denotes a selection operation on collection C, where $x = \bigwedge_T$ is a conjunction of selection predicates; each selection predicate $t \in T$ is in the form (a, ω, v), where a is a primitive attribute, $\omega \in \{=; >; <; \neq; \geq; \leq\}$ and v is a value.
$\pi_Y(C)$	Denotes a projection operation on collection C, where Y is a set of projection predicates; each projection predicate $y \in Y$ is in the form $y = \bigvee_A /f$ where A is a set of primitive attributes (of which the first non-null values is taken), and $/f$ indicates that the resulting attribute is named after feature f. It is $attr(f) \supseteq A$.
$\gamma_{(F',Z)}(C)$	Denotes an aggregation operation on collection C, where F' is the group-by set (i.e., a set of features) and Z is the set of aggregations; each aggregation is in the form (f, op) where f is a feature and op an aggregation function.
$(C_1) \cup (C_2)$	Denotes a union operation between collections C_1 and C_2.
$(C_1) \sqcup_{(a_i,a_j)} (C_2)$	Denotes a merge operation between collections C_1 and C_2 based on the equivalence $a_i = a_j$, with $(a_i, a_j) \in (C_1, C_2)$. See Definition 4.2.

R_{ij} results in a full-outerjoin between R_i and R_j where the couples of attributes linked by a mapping are merged through function $\Lambda\!\!\Lambda$. In particular, given a record $r \in R_{ij}$ obtained by joining $s \in R_i$ and $t \in R_j$ (i.e., $s[a_i] = t[a_j]$), then $\forall\,(a, a') \in (S_i, S_j) : a \equiv a'$ it is $r[rep(a)] = \Lambda\!\!\Lambda(s[a], t[a'])$.

Example 4.3 With reference to Table 1, let $S_1 \sqcup_{f_1} S_{10}$, $s \in C_6$ with schema S_1, $t \in C_4$ with schema S_{10}, $s[a_4] = t[a_6]$ where $attr(f_1) \supset \{a_4, a_6\}$. Let the values of ProductName be $s[a_7]$ = "Blueseventy Vision Goggles" and $t[a_9]$ = "B70 VG". The merge of s and t produces a record r where $r[a_7] = \Lambda\!\!\Lambda'(s[a_7], t[a_9])$ and $\Lambda\!\!\Lambda'$ is a conflict-resolution function that decides between "Blueseventy Vision Goggles" and "B70 VG" and produces a consistent result to answer the query in Example 4.1.

4.2 The query plan

Building the execution plan of a query first requires identifying the entities that need to be accessed, which are not limited to those containing the features selected in the query. For instance, a query asking for the average price of the items ordered by a customer requires to access not only entities

Customer and Orderline but also Order, even if no feature belonging to Order is mentioned in the query. Thus, we define the *query graph* as the subgraph of the entity graph that includes all and only the entities that need to be accessed to answer a certain query.

Definition 4.3 (Query graph) The query graph $G^{\mathcal{E}}_q$ is a subgraph of $G^{\mathcal{E}}$ (i.e., $G^{\mathcal{E}}_q = (\mathcal{E}_q \subseteq \mathcal{E}, L^{\mathcal{E}}_q \subseteq L^{\mathcal{E}})$) such that:

(i) $G^{\mathcal{E}}_q$ is minimally connected;
(ii) $\mathcal{E}_q \supseteq attr(q)$;
(iii) $\exists\, E^* \subseteq \mathcal{E}_q : E^* \supseteq q_\gamma,\ \forall\, E' \in \mathcal{E}_q$ it is $E^* \Rightarrow E^{'}$.

Condition (i) ensures that no unnecessary entity is accessed. Condition (ii) ensures that all attributes belonging to the features involved in the query are covered by the entities in $\mathcal{E}_q$. Condition (iii) entails the *compliance* of query q with the GPSJ semantics, that is, there exists an entity representing the events at the finest level of granularity (i.e., $E^* \Rightarrow E^{'}$ indicates that a directed path exists from E^* to every other entity $E^{'} \in \mathcal{E}_q$). Many subgraphs could exist for a given query since many-*-to-one* paths could exist, each associated with different semantics (e.g., an entity of sales could be associated with an entity of dates through the mappings on both *date of sale* and *date of shipping*). In this case, we rely on a user interaction to identify the adequate subgraph.

The query graph $G^{\mathcal{E}}_q$ is the starting point to define the execution plan in NRA for query q, i.e., the query plan P_q (an example is shown in Fig. 10).

Definition 4.4 (Query plan) A query plan is an NRA tree where the leaves denote an access to a collection (CA) and the root is either an aggregation (γ) or a projection (π).

As shown in Fig. 10, the leaves of the query plan can be organized into *entity plans*, and the leaves of each entity plan can be organized into *collection plans*. In the following paragraphs, we describe the top-down decomposition of query, entity, and collection plans, and the procedure to obtain them from the query graph. Such a procedure embeds a series of optimization techniques, which we highlight in Sect. 4.3.

4.2.1 Building the query plan

The rationale of the query plan is to first reconcile the records belonging to the same entity, and then join them with records from other entities; this is consistent with [16,19,20], where schemas of the same entity are joined together before being joined with schemas of different entities. Thus, a query plan is actually composed of one or more entity plans, which are merged through operator $\sqcup$.

The query plan P_q is organized as a left-deep tree of entity plans, where the order of the merge operations is optimized through a minimum selectivity heuristic [21]. Algorithm 1 incrementally produces P_q. The entities identified by the query graph (i.e., $\mathcal{E}_q$) are sorted based on the adopted heuristic in Line 1. Then, the plan is built as a left-deep tree by first defining the entity plan of the first entity (Line 3), and then progressively merging the entity plans of the subsequent entities (Lines 5 to 10). The function $createEntityPlan$ (Lines 3 and 7) is defined in Algorithm 2. The function $getLink$ (Line 8) retrieves from the query graph $G^{\mathcal{E}}_q$ the link l that connects the current entity E with those previously merged, i.e., $mergedEntitied$; this is necessary to identify the feature for the merge operation, indicated with $feat(l)$ (Line 9). Once every entity plan has been merged into a single NRA tree, the final operators (to be added as the root of the query plan) depend on the formulated query. If the query specifies an aggregation (i.e., $q_\gamma \neq \varnothing$), an aggregation operation is added as the root of the query plan (Lines 11 to 13); otherwise, a simple projection is added as the root of the query plan (Lines 14 to 16).

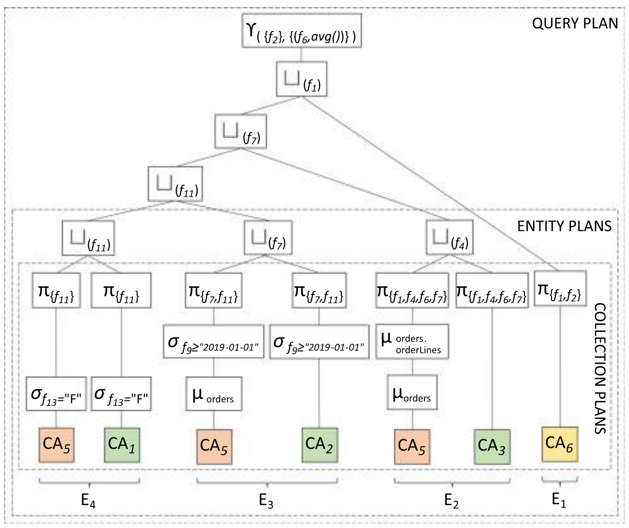


Fig. 10 The execution plan for the query in Example 4.1

Algorithm 1 Definition of the query plan P_q for query q.

```
INPUT $q = (q_\pi, q_\gamma, q_\sigma)$: a query; $G^{\mathcal{E}}_q = (\mathcal{E}_q, L^{\mathcal{E}}_q)$: the query graph.
OUTPUT $P_q$: the NRA query plan of $q$.
1: $entityList \leftarrow sortEntities(G^{\mathcal{E}}_q, q_\sigma)$   ▷ Apply minimum selectivity heuristic to sort entities
2: $E \leftarrow pop(entityList)$   ▷ $pop()$ extracts the first element of the list
3: $P_q \leftarrow createEntityPlan(q, E)$
4: $mergedEntities \leftarrow E$
5: while $entityList \neq \varnothing$ do
6:     $E = pop(entityList)$
7:     $rightPlan \leftarrow createEntityPlan(q, E)$
8:     $l \leftarrow getLink(G^{\mathcal{E}}_q, mergedEntities, E)$
9:     $P_q \leftarrow extendPlansWithBinaryOp(P_q, rightPlan, \sqcup, feat(l))$
10:    $mergedEntities \leftarrow mergedEntities \cup E$
11: if $q_\gamma \neq \varnothing$ then   ▷ Groups $q_\gamma$ by $q_\pi$
12:     $predicate \leftarrow (q_\pi, q_\gamma)$
13:     $P_q \leftarrow extendPlanWithUnaryOp(P_q, \gamma, predicate)$
14: else   ▷ Projects on $q_\pi$
15:     $predicate \leftarrow q_\pi$
16:     $P_q \leftarrow extendPlanWithUnaryOp(P_q, \pi, predicate)$
17: return $P_q$
```

Algorithm 2 createEntityPlan

```
INPUT $q = (q_\pi, q_\gamma, q_\sigma)$: a query; $E$: an entity; $\mathcal{S}_E^q$.
OUTPUT $P_E$: the NRA entity plan for $E$.
1: $\mathcal{S}_E^q = \bigcup_{S \in \mathcal{S}_E} S \cap feat(q_\sigma) \neq \varnothing$   ▷ The schemas of $E$ that need to be accessed
2: $collections \leftarrow \bigcup_{S \in \mathcal{S}_E^q} col(S)$
3: $collectionList \leftarrow sortCollection(collections, q_\sigma)$   ▷ Apply min.sel. heuristics to sort collections
4: $col \leftarrow pop(collectionList)$
5: $P_E \leftarrow createCollectionPlan(q, \mathcal{S}_{col}^q)$
6: $f \leftarrow feat(\mathcal{S}_{col}^q)$   ▷ Get the feature $f$ that represents the keys of the schemas
7: while $collectionList \neq \varnothing$ do
8:     $col \leftarrow pop(collectionList)$
9:     $right \leftarrow createCollectionPlan(q, \mathcal{S}_{col}^q)$
10:    if $\phi_E = \texttt{true}$ then
11:        $P_E \leftarrow extendPlansWithBinaryOp(P_E, right, \sqcup, f)$
12:    else
13:        $P_E \leftarrow extendPlansWithBinaryOp(P_E, right, \cup, f)$
14: return $P_E$
```

The two functions $extendPlanWithUnaryOp$ and $extendPlansWithBinaryOp$ (lines 9, 13, and 16 in Algorithm 1), respectively, extend the existing plan with a new unary or binary operation; naturally, the former requires in input a single plan (to be extended with a unary operation), while the latter requires two plans to be merged (either through a merge or union operation).

4.2.2 Building an entity plan

Similarly to the query plan, an entity plan is a left-deep tree where the leaves are collection plans. The goal of the entity plan is to merge the records obtained from its schemas. However, current NoSQL technologies do not allow access to collections' records based on a certain schema (collections are schemaless by definition); this is why we define the leaves as collection plans instead of schema plans. The order of the merge operations between collection plans is determined by adopting the same heuristics.

Algorithm 2 incrementally produces the entity plan P_E for a given entity E. Let $\mathcal{S}_E^q = \bigcup_{S \in \mathcal{S}_E} S \cap feat(q_\sigma) \neq \varnothing$ be the set of schemas belonging to E that need to be accessed: in particular, we can exclude the schemas that do not contain an attribute for the features in q_σ, because the filter would automatically discard every record. To define collection plans, we identify the distinct set of collections that need to be accessed in Line 2, then we sort them based on the adopted heuristic in Line 3. The entity plan is built as a left-deep tree by first defining the collection plan of the first collection (Lines 4, 5), and then progressively merging the collection plans of the subsequent collections (Lines 7 to 13); the function $createCollectionPlan$ is defined in Algorithm 3. Remarkably, collection plans are merged with $\sqcup$ only if E suffers from record overlapping; otherwise, a simple (and less costly) union operation is sufficient to put together the records from each collection plan.

4.2.3 Building a collection plan

Finally, each collection plan describes the sequence of unary NRA operations to collect the records of a certain entity E in a collection col. Since the collection may contain several schemas belonging to the same entity, the collection plan takes into consideration the inherent schema variations: given $\mathcal{S}_E^q$ the set of schemas of E that need to be accessed, we refer to $\mathcal{S}_{col}^q \subseteq \mathcal{S}_E^q$ as the subset of schemas to be considered for collection col.

Algorithm 3 produces the collection plan P_{col} by taking into consideration the schema variety within col. The collection plan is defined as an ordered sequence of unary NRA operations in the following order: optional unnesting operations, an optional selection operation, and a final projection operation. We remark that such order is the most obvious one, as (i) unnesting is necessary to first unveil the nested attributes, and (ii) it is usually a good practice to apply selection predicates as soon as possible [22]. The plan P_{col} is built bottom-up as follows.

- The first operation is the collection access CA to col (Line 2).
- Unnesting operators are possibly added (Lines 3 to 8) in case one or more schemas are nested within arrays (i.e., $|S^\mu| \geq 1$). A simple check for duplicates is done in Line 6 in case $\exists\ (S_1, S_2) \in \mathcal{S}_{col}^q : S_1^\mu \cap S_2^\mu \neq \varnothing$; unnesting operations are added to P_{col} in Line 8.
- The optional selection operation is built in Lines 9 to 14. For each feature that needs a selection, we build a disjunction of predicate that considers every schema variation of f (Line 12); for instance, a selection $(f_2, =, v)$ (where v is some value) translates to a selection $(a_7, =, v) \vee (a_8, =, v) \vee (a_9, =, v)$. Then, the final selection predicate is the conjunction ($\wedge$) of the predicates built for each feature (Line 14).
- Finally, the projection operation is built in Lines 15 to 20. Let $F_\pi = \{feat(q_\pi) \cup feat(q_\gamma) \cup F_\sqcup\}$ (used in Line 10) be the set of features to be projected, where $\forall\ l \in L_q^{\mathcal{E}}$ it is $F_\sqcup = feat(l)$ is the set of features whose attributes are necessary for merge operations. For each feature $f \in F_\pi$ representing attributes in $\mathcal{S}_{col}^q$ we project a single attribute (named after $rep(f)$) that contains the only non-null value among its schema variations (simplified in Line 19 as a disjunction over each $a \in A$). We remark that, at this stage, we also apply the transcoding functions φ in order to consistently compare record values in the merge operations that will follow.

Example 4.4 Figure 10 shows the query plan of the query from Example 4.1.

Algorithm 3 createCollectionPlan

```
INPUT $q = (q_\pi, q_\gamma, q_\sigma)$: a query; $\mathcal{S}^q_{col}$: the set of schemas of a certain collection $col$ involved in $q$.
OUTPUT $P_{col}$: the NRA collection plan of $col$.
1: $P_{col} \leftarrow$ new $PlanNode()$
2: $P_{col} \leftarrow extendPlanWithUnaryOp(P_{col}, \mathsf{CA}, col)$    ▷ Start with the collection access
3: $predicateSet \leftarrow \varnothing$
4: for all $S \in \mathcal{S}^q_{col}$ do
5:     for $i = 1$ to $|S^\mu|$ step 1 do
6:         if $predicateSet \cap S^\mu[i] = \varnothing$ then    ▷ Duplicates check
7:             $predicateSet \leftarrow predicateSet \cup S^\mu[i]$
8:             $P_{col} \leftarrow extendPlanWithUnaryOp(P_{col}, \mu, S^\mu[i])$    ▷ Add optional unnesting ops
9: $predicateSet \leftarrow \varnothing$
10: for all $f \in q_\sigma$ do
11:     $A \leftarrow \mathcal{S}^q_{col} \cap f$
12:     $predicateSet \leftarrow predicateSet \cup (\bigvee_{a \in A}(\varphi_{(rep(f),a)}(a), \omega, v))$
13: if $predicateSet \neq \varnothing$ then
14:     $P_{col} \leftarrow extendPlanWithUnaryOp(P_{col}, \sigma, \bigwedge_{p \in predicateSet})$    ▷ Add optional selection op
15: $predicateSet \leftarrow \emptyset$
16: for all $f \in F_\pi = \{feat(q_\pi) \cup feat(q_\gamma) \cup F_{\sqcup}\}$ do    ▷ The set of features to be projected
17:     $A \leftarrow \mathcal{S}^q_{col} \cap attr(f)$
18:     if $A \neq \varnothing$ then
19:         $predicateSet \leftarrow (\bigvee_{a \in A} \varphi_{(rep(f),a)}(a)) \,/\, rep(f)$
20: $P_{col} \leftarrow extendPlanWithUnaryOp(P_{col}, \pi, predicateSet)$    ▷ Add projection op
21: return $P_{col}$
```

4.3 Optimizations

The distributed and multi-model nature of the multistore environment, coupled with the high-variety scenario covered in this paper, offers several opportunities for the optimization of query plans. The approach described so far already adopts a set of optimization techniques to produce a refined execution plan.

- **Schema plan grouping**. Since we model several schemas within the same collection, the naive way would be to produce a query plan with as many leaves (i.e., collection accesses) as the number of schemas. As described in Sect. 4.2, we optimize it in order to have as many leaves as $|\mathcal{E}^q| \cdot |col(E)|$, where $|\mathcal{E}^q|$ is the number of entities in the query, and $|col(E)|$ is the number of collections for an entity $E \in \mathcal{E}^q$. Thus, the collection plan exploits mappings to query several schemas in a single pass. This is evident in Algorithm 3, where we identify $\mathcal{S}^q_{col}$ as the subset of schemas to be considered for the plan of collection col. In particular, $\mathcal{S}^q_{col}$ is used in Lines 4, 11, and 17 to, respectively, define unnesting, selection, and projection operations on col.
- **Predicate push-down**. This is one of the most basic optimization techniques, which consists of applying selection predicates as close to the source as possible. We apply them in the collection plan (Algorithm 3, Lines 9 to 14) right after unnesting the necessary arrays (i.e., before any projection, merge, and aggregation operation).
- **Merge sequence reordering**. When the query involves three or more collections, the order in which collections are merged together has an impact on performance. In this work, we rely on a minimum selectivity heuristics [21] to determine the order of merge operations. The basic idea is to start from the one with the lowest cardinality and progressively merge it with collections with increasing cardinality. This technique is used to decide the join sequence of collection plans within a single entity plan (Line 3 in Algorithm 2) and the join sequence of entity plans within the query plan (Line 1 in Algorithm 1). Notice that the reordering of entity plans within query plans considers the former as atomic blocks of operation, i.e., when a reordering takes place, the inner structure of entity plans remains unchanged; the same principle applies to the reordering of collection plans within entity plans. With reference to Algorithm 1, let $E_i \in \mathcal{E}_q$ be the entity with the smallest cardinality and $\mathcal{E}'_q$ the set of entities directly connected to E_i in $L^{\mathcal{E}}_q$; then E_i is merged with $E_j \in \mathcal{E}'_q$ whose cardinality is the smallest one. The same step is repeated (at the second iteration, $\mathcal{E}'_q$ is the set of entities directly connected to either E_i or E_j) until all entities in $\mathcal{E}_q$ have been merged. To estimate entities' cardinalities we take into consideration the selection predicates in q_σ; in turn, this requires collecting statistics from the databases. The literature on such topics is very broad. The accuracy of the estimate strictly depends on the collected information and the assumptions made on data distribution. Following several query cost models, in this paper we assume uniformity of attribute values, attribute values independence, and join containment.
- **Column pruning**. This technique consists in extracting from each collection the only attributes corresponding to features that are relevant for the query, i.e., those required by the final projection (or aggregation) operation and those necessary for merge operations. We refer to these features as F_π in Algorithm 3, Line 10. By keeping only the minimum set of attributes we minimize the amount of data that needs to be moved across the network. We finally remark that column pruning is also adopted after each merge operation to prune join attributes that are not needed anymore (although this is not shown in Algorithms 1 and 2 for simplicity).

Although some of the mentioned optimizations are not new to DBMSs and execution engines, their application in a complex multistore environment is not straightforward. Apache Spark (i.e., the one we use in our prototype) uses Catalyst to provide optimization techniques in query executions. However, Catalyst is not aware of the constraints that guarantee the correctness of the query plan and, ultimately, of the query result. As explained above, our heuristics preserves the inner structure of entity plans when reordering them within a query plan (and similarly for collection plans reordering within an entity plan). Since Catalyst has no notion of the internal organization of the query plan, its reordering strategy may swap operations that break the boundaries of collection or entity plans (e.g., a collection plan may be moved to a different entity plan), thus compromising the correctness of the result. For this reason, these optimization routines are directly defined within our approach.

5 Experimental evaluation

In this section, we discuss our experiments that evaluate the performance of our approach from several perspectives.

5.1 The prototypical setup

Our reference architecture is a two-rack Big Data cluster of 18 Ubuntu machines with a minimum configuration of i7 8-core CPU @3.2GHz, 32GB RAM, and 6TB hard disk drives. Each machine runs the Cloudera Distribution for Apache Hadoop (CDH) 6.2.0. The multistore implementation relies on PostgreSQL, MongoDB, Cassandra, and Redis as relational, document-based, wide-column, and key-value DBMSs, respectively. PostgreSQL is installed on a single machine, while NoSQL stores are distributed across 15 machines. The algorithmic implementation of the approach is based on Apache Spark, i.e., one of the most used open-source execution frameworks for Apache Hadoop clusters; it provides connectors to most DBMSs, including those in our multistore.

Figure 11 provides an overview of our prototypical implementation from a functional and technological perspective. The main application modules (i.e., the query planner and the dataspace manager) are written in Scala, and they are coupled with an HTTP server that enables user interactions through REST APIs. The dataspace manager includes functionalities to build, update, and visualize the content of the dataspace (whose metadata are stored in the same PostgreSQL instance used for the data), while the query planner implements the algorithms (i.e., Algorithms 1 to 3) and the optimization techniques (see Sect. 4.3). Queries are formulated by relying on the SQL APIs exposed by Spark's DataFrame abstraction; the new merge operator fits this abstraction: it is implemented as a full-outerjoin between two DataFrames, on top of which are applied custom User Defined Functions (UDFs) representing the conflict-resolution functions.

The query execution framework consists of 8 executors, each with 6 CPU cores and 8GB RAM[5]. The data is collected from the underlying DBMSs by pushing to the latter as much computation as possible; then, the Query execution framework runs in-memory computation to complete the execution of the query and obtain the final results, that are finally returned to the user.

[5] The number of resources allocated to the Query execution framework is limited to grant enough resources to the underlying DBMSs and other concurrent research applications running on our cluster.

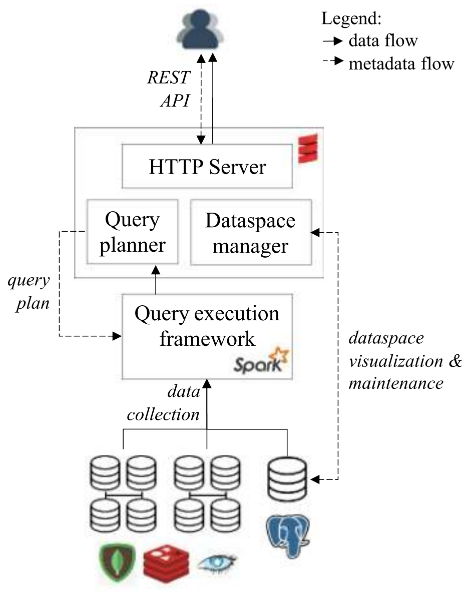


Fig. 11 Functional and technological overview of the approach

To evaluate the approach in terms of scalability, we have implemented the multistore in four scale factors, i.e., 1, 10, 100, and 1000. The size of each collection in the different scale factors is reported in Table 3. We recall from Sect. 2 that there is a 20% overlap of customers between C_1 and C_5, and a 60% overlap of products between C_4 and C_6. The number of products is fixed in each implementation, together with the ratio of orders per customer (i.e., 15 on average) and the ratio of order lines per order (i.e., 5 on average); what scales is the number of customers and, consequently, the overall number of orders and order lines. Table 3 also reports the partitioning key of each collection; whereas partitioning (i.e., sharding) records is a necessity in distributed DBMSs, it also helps increasing query efficiency in presence of certain selection predicates in every DBMS. Notice that C_4 in partitioned only on a customer attribute because orders and order lines are nested within customers.

As we recall from Sect. 3.3, the preparation of the dataspace is done in three steps.

– Schema extraction is run in parallel on every DBMS. Its execution time ranges from few seconds to up to thirty minutes, depending on the considered scale factor. This is compatible with execution times from related works on schema extraction [23,24]. We remark that extracting schemas from non-relational collections requires a full

Table 3 Number of records and partitioning key for each collection in the different scale factors

Entity	Coll.	SF1	SF10	SF100	SF1000	Part. key
Customer	C_1	9K	90K	900K	9M	FirstName
	C_5	3K	30K	300K	3M	FirstName
	Total	**10K**	**100K**	**1M**	**10M**	–
Order	C_2	50K	500K	5M	50M	OrderDate
	C_5	15K	150K	1.5M	15M	–
	Total	**65K**	**650K**	**6.5M**	**65M**	–
Orderline	C_3	230K	2.3M	23M	230M	OrderId
	C_5	60K	600K	6M	60M	–
	Total	**290K**	**2.9M**	**29M**	**290M**	–
Product	C_4	8K	80K	800K	8M	ProductName
	C_6	8K	80K	800K	8M	–
	Total	**10K**	**100K**	**1M**	**10M**	–
Invoice	C_7	**65K**	**650K**	**6.5M**	**65M**	OrderId

scan of the latter, as most NoSQL stores have no schema definition for collections: naively, a schema is generated for each record, but only distinct schemas are kept. The efficiency of this task could be improved by adopting approximation techniques (e.g., sampling to avoid a full scan of every collection) and an incremental strategy (i.e., to consider only new/updated records); nonetheless, the optimization of the schema extraction task is out of the scope of this paper.

– The definition of mappings is done manually in our case study; although it could be made automatic by implementing some schema matching algorithm [17] or by embedding existing tools (e.g., Coma 3.0 [18]), it is out of the scope of this paper to optimize this step.
– Features and entities are automatically inferred from the mappings and the one-to-one relationships between schemas, respectively; the execution time of this step is almost immediate. The recognition of which entities suffer from record overlapping (i.e., setting ϕ_E for each E) is done manually in our case study, but it could be made automatic by implementing a procedure that compares key values in the schemas and looks for matches that reveal an overlap.

Ultimately, the dataspace's metadata occupy less than 100 kB in every scale factor.

5.2 Scalability under data variety

The first experiments are aimed at assessing the scalability of the system under different levels of variety in the data. In particular, we measure how the query planner and the merge operator perform by varying the number of schemas and the amount of overlapping records, respectively.

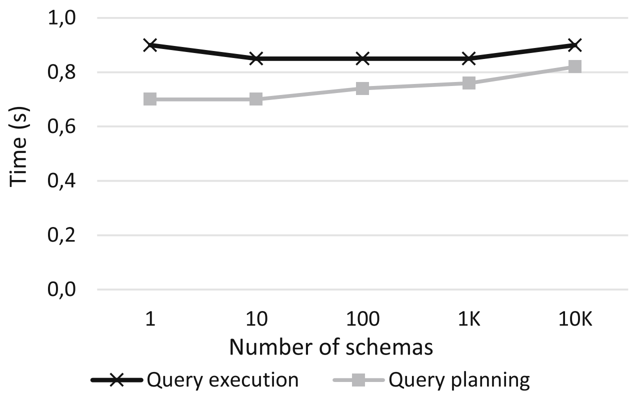


Fig. 12 Query planning and execution time by varying the number of schemas

To evaluate the query planner, we build two synthetic collections of customer records, each with 50000 records; one is stored on the relational database with a single schema, the other on the document-based database with a varying number of schemas, from 1 to 10000. The latter is a borderline scenario, as (from our experience) collections with high variety rarely exceed the hundreds of schemas. Figure 12 shows execution times (averaged from 5 executions) of a query that merges and aggregates the data from both collections; we consider a single-core Spark instance of the middleware, so as to exclude variations due to parallelization. The results show that both query planning and execution are not affected by the number schemas, as minimum oscillations are observed. This is expected for the query execution, since resolving schema heterogeneity consists of low-impact operations such as renaming attributes' names. As to query planning, even though the complexity of the procedure is linear with the number of schemas, the cardinality of the latter is not sufficient to impact the overall planning time. Ultimately,

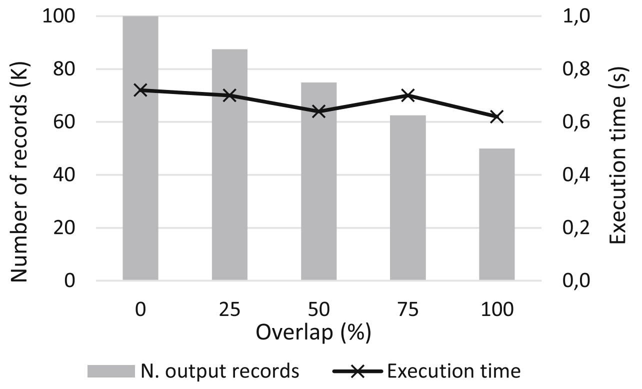


Fig. 13 Performance of the merge operator with varying levels of record overlapping

this proves a good efficiency of the query planner in handling high levels of schema heterogeneity.

As to the merge operator, we measure its performance under varying levels of record overlapping. Starting from the two previous collections of customer records, we remove schema heterogeneity and progressively increase the level of overlap between the records from 0% to 100%. The results are shown in Fig. 13; the execution times (averaged from 5 executions) correspond to the single merge operation (i.e., the two read operations are not considered). By increasing the level of overlap, the merge operation naturally returns a progressively lower amount of records; nonetheless, the performance of the merge operator is not influenced by this factor (the observed variations are minimal). This behavior is expected, as the complexity of the merge operation is the same as a full-outerjoin operation and the conflict-resolution functions are not computationally expensive.

5.3 Efficiency evaluation

The workload we devise consists of 48 GPSJ queries that vary in terms of group-by set strength, selection predicate selectivity, and the number of entities involved (i.e., the size of the query graph).

- The group-by set is either absent (i.e., only a simple projection is carried out, without aggregation), weak (i.e., it involves features with high cardinality, resulting in several groups), or strong (i.e., it involves features with low cardinality, resulting in few groups). This parameter affects the cardinality of the results, which (on average) is below 10^5 when the group-by set is absent, 10^4 when it is weak, and 10^2 when it is strong.
- The selection predicate is either absent, weak (i.e., its selectivity is low), or strong (i.e., its selectivity is high). This parameter affects the number of records involved in the queries, which is between 80% and 40% in weak selections, and between 5% and 0.01% in strong selections.
- We devise 6 different query graphs, varying the number of entities involved in the query (i.e., $|\mathcal{E}_q|$) from 1 to all 5 of them.

Table 4 Average execution times of the workload queries, by varying the group-by set's strength

GB set	Execution times (s ± RSD)			
	SF 1	SF 10	SF 100	SF 1000
Absent	$1.2 \pm 42\%$	$4.0 \pm 45\%$	$18.0 \pm 38\%$	$559.2 \pm 79\%$
Weak	$1.5 \pm 40\%$	$5.0 \pm 52\%$	$28.4 \pm 54\%$	$603.7 \pm 90\%$
Strong	$1.5 \pm 40\%$	$4.6 \pm 48\%$	$26.1 \pm 54\%$	$593.8 \pm 84\%$

This determines a total of 54 combinations; however, queries with no group-by set and no/weak selection predicates (i.e., non-analytical queries) are hardly applicable in large query graphs, where the cardinality of the result would be close to the size of the entire database. Thus, we exclude these two kinds of queries on the three largest query graphs, obtaining a total of 48 queries. The detailed list of queries is provided as Supplementary Information with the paper.

Execution times and scalability. The workload queries have been executed on the multistore against every scale factor. The execution time (always obtained as the average of 5 executions) mainly depends on the complexity of both the query and the dataset, but it is also affected by the way the computing resources have been allocated on the cluster. Big Data frameworks like Spark try to honor the locality principle, but they do not guarantee that the computation always happens on the same nodes; thus, execution times of the same computation may vary depending on the amount of data shuffling required when the locality principle is not met. Conversely, the time taken to build the execution plan (i.e., by running Algorithms 1–3) is affected by neither the query and dataset complexity (as shown in Sect. 5.2) nor the Big Data framework (as the implementation is centralized), and it always performs in sub-second times.

The query execution times (in seconds, together with the relative standard deviation (RSD)) are shown in Tables 4, 5, and 6; each table shows average times by, respectively, grouping the workload queries by group-set strength, selection predicate strength, and the number of entities in the query. Times increase as expected with the scale factor (especially evident when moving from SF 10 to 100), while selection predicates appear to have little effect. Indeed, the system can exploit local indexing and/or partitioning only on the collections on which the selection predicates are applied. The behavior under different group-by conditions is also different: execution times are faster in absence of group-by set because no data shuffling is required to carry out an aggrega-

Table 5 Average execution times of the workload queries, by varying selection predicates' strength

Selection predicate	Execution times (s ± RSD)			
	SF 1	SF 10	SF 100	SF 1000
Absent	1.5 ± 40%	5.0 ± 50%	31.0 ± 55%	588.1 ± 88%
Weak	1.6 ± 38%	5.1 ± 51%	29.0 ± 53%	562.7 ± 87%
Strong	1.3 ± 38%	4.2 ± 45%	20.6 ± 43%	569.2 ± 82%

tion; when the aggregation is necessary, the system performs slightly better if the group-by set is stronger, where fewer records are generated and shuffled. This is due to the usage of combining strategies that carry out map-side aggregation, thus shuffling less records for the reduce-side aggregation.

Local vs middleware computation. A second evaluation is made to compare the amount of computation assigned to the source against the one assigned to the middleware. For each query execution, we consider the local computation as the sum of the execution times of Spark's tasks in charge of reading from the DBMSs, and middleware computation as the sum of the execution times of Spark's remaining tasks[6]. The results are shown in Fig. 14. Interestingly, the percentage of computation demanded from the middleware decreases with the increase in the scale factor. In absolute terms, the local computation demanded from the sources scales linearly with the scale factor, while the middleware computation initially scales sublinearly (about 2x from SF 1 to SF 10, about 5x from SF 10 to SF 100). This is due to the middleware suffering the distributed framework's overhead in handling low amounts of data in the smaller scale factors. Ultimately, we infer that relying on middleware for joining and merging records (which involves shuffling data on the network between different software tools) does not have a major impact, especially when the amount of data to be considered becomes larger.

Optimization impact. Finally, we measure the impact of our optimization techniques by selectively switching them off and verifying the execution times. We specifically focus on the schema plan grouping (SPG), merge sequence reordering (MSR), and column pruning (CP) optimizations. In this case, we obtain the measurements for each query and evaluate, on each scale factor, the average loss in percentage with respect to the execution with every optimization enabled. The results are shown in Table 7. While the contribution of CP is limited and erratic, SPG emerges as the optimization producing the most significant advantage. This is expected, as turning it off means issuing several queries on the same collections, which clearly has a major impact—especially with increasing scale factors, where the weight of the local computation is higher (as seen in the previous evaluation). MSR is also quite relevant; unlike SPG, MSR's contribution is decreasing with the scale factor, since the weight of the middleware computation decreases as well; the only exception is in SF1, where the benefit of MSR in queries with low execution times is mitigated by the distributed framework's overhead.

5.4 Effectiveness evaluation

Adopting a pay-as-you-go approach entails that query answer quality depends on the number of defined mappings. In this subsection, we analyze how the results vary by selectively removing some of the mappings. Our goal is to quantify the impact of mappings in producing a correct result and to demonstrate the issues that would arise by adopting a system that does not entail a mechanism to solve schema heterogeneity and record overlapping.

Let $\mathcal{D}^*$ be the ground-truth dataspace (i.e., the one with all mappings identified); we consider three different scenar-

[6] By focusing on tasks' execution times, we avoid taking into account the framework's parallelization.

Table 6 Average execution times of the workload queries, by varying the number of entities in the query

$\|\mathcal{E}_q\|$	Execution times (s ± RSD)			
	SF 1	SF 10	SF 100	SF 1000
1	0.4 ± 0%	0.7 ± 14%	5.1 ± 12%	18.2 ± 14%
2	1.2 ± 8%	3.5 ± 9%	18.2 ± 15%	155.2 ± 12%
3	1.5 ± 13%	5.0 ± 8%	27.5 ± 26%	615.2 ± 24%
4	1.9 ± 11%	6.1 ± 13%	35.3 ± 29%	1072.1 ± 17%
5	2.2 ± 14%	7.8 ± 19%	42.2 ± 30%	1114.4 ± 16%

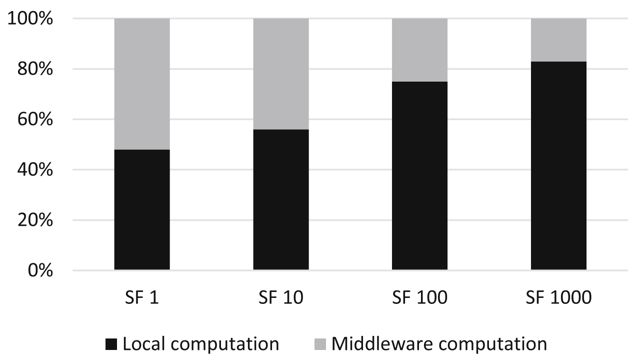


Fig. 14 Comparison of average local and middleware computation on all queries with different scale factors

Table 7 Average increase in query execution times by switching off schema plan grouping (SPG), merge sequence reordering (MSR), and column pruning (CP) optimizations

Optimizations turned off	Execution times increase (% ± SD)			
	SF 1	SF 10	SF 100	SF 1000
SPG	140 ± 30	200 ± 100	290 ± 170	350 ± 200
MSR	27 ± 23	40 ± 35	16 ± 8	2 ± 1
CP	2.4 ± 2	6.1 ± 5	3.0 ± 2	4.2 ± 3

ios, each represented by a different dataspace (i.e., $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$), where different types of mappings have been selectively removed with respect to $\mathcal{D}^*$ (we refer the reader to Table 1 for attributes' and features' definitions in our case study). Table 8 summarizes the characteristics of each scenario and measures the average quality degradation of those workload queries that are affected by the removal of the mappings. Inspired by [19], Table 9 evaluates the following.

- *Query density* as the percentage of non-null cells in the query results.
- *Query coverage* as the percentage of records considered by the query with respect to $\mathcal{D}^*$.
- *Aggregation veracity*, i.e., whether the aggregation of partial results where mappings are missing is consistent with the results obtained in the ground-truth dataspace.
- *Selection support*, i.e., whether the absence of mapping hinders the capability of applying selection predicates.

Simple attributes. In $\mathcal{D}_1$ we consider a lack of mapping between attributes within the same entity; for instance, the attributes representing the OrderDate of the Order are not reconciled by a mapping, thus $a_{29} \not\equiv a_{30}$. Failing to recognize this kind of mapping means that each attribute gets represented by a distinct feature (e.g., $f_9^{'}$ in C_2 and $f_9^{''}$ in C_5) and must be queried separately. This scenario has an impact on the query results in terms of density (i.e., the percentage of non-null values), meaning that:

- In case of projections, each feature returns a null value for every record in which the respective attribute is not defined (e.g., approximately 50% of null values are returned by both $f_9^{'}$ and $f_9^{''}$). Notice that actual query densities in Table 8 are higher due to the projection of other features without null values.
- In case of selection predicates, a disjunction of separate conditions would need to be manually formulated by the user on each feature (e.g., $f_9^{'} <$ "2020-01-01" $\vee f_9^{''} <$ "2020-01-01"); however, this would not be answerable, as we currently support only conjunctions of selection predicates.

Simple attributes with record overlapping. In $\mathcal{D}_2$ we suppose that the same scenario in $\mathcal{D}_1$ applies to attributes of collections with overlapping records; for instance, the attributes representing the LastName of Customers are not reconciled by a mapping, thus $a_{36} \not\equiv a_{37}$. Failing to recognize this kind of mapping means not only that (i) as in $\mathcal{D}_1$, each attribute gets represented by a distinct feature (e.g., $f_{12}^{'}$ in C_1 and $f_{12}^{''}$ in C_5), but (ii) it also introduces a problem in terms of veracity of the results. When records are overlapping, any potential conflict (e.g., different last names found in different records of the same customer) are solved by the merge functions $\bigwedge\!\!\!\bigwedge$ defined in the features in $\mathcal{D}^*$. In $\mathcal{D}_2$, having distinct features means that the respective attributes can be queried separately, but the obtained results cannot be easily merged. For instance, consider two queries that sum the TotalPrice by LastName, i.e., $q' = (\{f_{12}^{'}\}, \{f_8, sum()\})$, and $q'' = (\{f_{12}^{''}\}, \{f_8, sum()\})$; an excerpt of the queries' results is shown in Table 9, together with the actual results from $\mathcal{D}^*$. Without record overlapping, the results from q' could have been summed to those from q'' to obtain the ground truth values. This is not necessarily true in presence of record overlapping, because the $\bigwedge\!\!\!\bigwedge$ function in $\mathcal{D}^*$ resolves conflicts in the last names before the aggregation and produces different results. In particular, we measured a $\pm 111\%$ difference between the sums of total prices obtained in $\mathcal{D}^*$ and those obtained in $\mathcal{D}_2$ by summing the results of q' and q''.

Key attributes. In $\mathcal{D}_3$ we consider a lack of mapping involving key attributes; for instance, a_{32} (i.e., the key of Customer in C_1) is not mapped to either a_{34} (i.e., the key of Customer in C_5), nor to a_{33} and a_{35} (i.e., the attributes in the Order referencing the key of the Customer). Failing to recognize this kind of mapping means that (i) as in $\mathcal{D}_1$ and $\mathcal{D}_2$, two features are created in $\mathcal{D}_3$ to represent the TaxId (e.g., $f_{11}^{'}$ and $f_{11}^{''}$), but also that (ii) two separate entities are defined to represent customers (e.g., Customer$^{'}$ and Customer$^{''}$), where only one of the two entities is actually linked to the Order. The main impact of this scenario on the query results is in terms of coverage [19] (i.e., the number of returned records), meaning that the records of the customer cannot be queried altogether. In particular:

- A query involving features of either one of the two entities will return only a selected number of records; thus, the queries will mostly have full density, but the coverage will decrease significantly.
- A query involving both features is not answerable, because they are not linked in the entity graph of $\mathcal{D}_3$.

Table 8 Evaluation of quality degradation under scenarios with selective mappings removed from the ground-truth dataspace $\mathcal{D}^*$. Query density and coverage are measured only on the workload queries actually affected by mappings removals

Dataspace	Removed mappings	Query density	Query coverage	Aggregation veracity	Selection support
$\mathcal{D}_1$	$a_{29} \not\equiv a_{30}$	70.3%	100%	Yes	Partial
$\mathcal{D}_2$	$a_{36} \not\equiv a_{37}$	80.0%	100%	No	Partial
$\mathcal{D}_3$	$a_{32} \not\equiv a_{33}$, $a_{32} \not\equiv a_{34}$, $a_{32} \not\equiv a_{35}$	91.1%	64.3%	Yes	Partial

Table 9 The absence of a mapping between the two LastName attributes in $\mathcal{D}_2$ does not trigger the conflict-resolution function between overlapping records and leads to inconsistent results, as the sums of the two partial results on $\mathcal{D}_2$ do not always match the true results on $\mathcal{D}^*$

$\mathcal{D}_2$				$\mathcal{D}^*$	
f'_{12}	$\{f_8, sum()\}$	f''_{12}	$\{f_8, sum()\}$	f_{12}	$\{f_8, sum()\}$
Faye	201542.1	Faye	194213.3	Faye	366440.2
Baloch	178805.2	Baloch	197430.7	Baloch	372510.7
Francois	54354.3			Francois	54354.3
Alschitz	11082.4	Alschitz	9030.1	Alschitz	20523.0
		Guelleh	67471.7	Guelleh	67471.7
		Akongo	186595.7	Akongo	186595.7
Nagy	118644.1	Nagy	136006.7	Nagy	289375.9

5.5 Comparison with related work

The novel scenario considered in this paper is the one where schema heterogeneity and record overlapping prevent users from directly issuing analytical queries over a multistore. In this section, we compare with alternative approaches by analyzing how the latter would tackle the same problem.

Reconciled level materialization. This is the classic Data Warehouse approach: a fully reconciled schema is created and loaded in batch mode via an ETL procedure [25,26]. Alternatively, a trigger-based approach can be adopted to feed the materialized view [27]. This solution favors the optimization of query time at the expense of making the system very rigid: (a) maintainability is affected, since every schema change entails an update of the ETL procedure; (b) the pay-as-you-go principle is compromised and a strong initial modeling effort is required; (c) materialized views and data sources are no more synchronized and the misalignment depends on how often the ETL procedure is executed. In our case study, the time to run a full materialization scales linearly with the scale factor and reaches up to 6 hours with SF 1000.

Multistore post-processing. The alternative solution is to rely on existing multistore approaches which enable cross-database querying through a common language or a mediating layer. In this case, the system supports data model heterogeneity, and there is no need to develop ETL procedures as querying would be carried out directly on the existing collections. However, existing systems do not support the resolution of schema heterogeneity and record overlapping that must be carried out a posteriori after having retrieved an intermediate result. Besides involving an extra human effort, this approach determines a higher computational cost since the intermediate data will necessarily be more numerous since, for example, filtering and grouping must be necessarily postponed. In particular, adopting this approach in our 48 queries benchmark requires the middleware to return a volume of data 12 times larger.

6 Related literature

The problem of querying distributed datasets has been considered by the community since the notion of the federated databases [28]. The variety in terms of available data models [29] (e.g., relational, wide-column, or document-oriented) responds to different requirements of modern data-intensive applications, but providing transparent querying mechanisms to query large-scale collections on heterogeneous data stores is an active research area [4]. In the following, we distinguish three main classes of solutions to resolve problems related to querying high-variety data: in Sect. 6.1, we discuss those addressing the presence of heterogeneous structures within the same data model; in Sect. 6.2, we discuss those addressing the querying problem across different data models; in Sect. 6.3, we discuss those addressing the resolution of record overlapping. Whereas all mentioned papers separately handle the different problems, to the best of our knowledge this is the first work to handle all of them.

6.1 Schema heterogeneity

Data model transformation. This class of work suggests performing data model transformation to facilitate the access to data having heterogeneous structures. The common strategy consists in changing the underlying data model, usually from a non-relational to a relational data model. This kind of solution leads to the loss of the schemaless flexibility guaranteed in most NoSQL stores in favor of the use of conventional relational querying and storing techniques. In particular, custom transformations and mappings are typically defined to move data from one data model to the other [25,26]. A mainstream approach widely used while dealing with heterogeneous XML databases is to transform documents into relation data model [30–32]. Other alternatives suggest storing documents on the wide-column data model. For instance, MonetDB [33] uses specialized data encoding, join methods, and storage for managing documents encoded in XML on the wide-column data model. In [30], the authors use the document type definition, i.e., DTD, to flatten documents and map documents into relational tables. However, despite the advantages of using relational schema and the expressiveness power of relational operators, partitioning data into tables by attributes [32] affects the performance of the relational system. This is due to the need of performing multiple joins to reconstruct the initial data. Furthermore, users of these systems have to learn new schemas every time new data are inserted (or updated) because it is necessary to re-generate the relational views. Another line of work introduces data model transformation between NoSQL stores. In [34], the authors introduce a tool-based advisor with a cost model dedicated to data migration scenarios. However, this approach mainly focuses on optimizing the costs related to migrating data from one data model to another, and it does not consider cross-data model querying nor resolving the problem of schema heterogeneity within the same data model.

Schema Versioning This class of work identifies the different co-existing versions within one database and adds a transparent layer to query tables having different representations for their schema regardless of the version used to formulate queries. This line of work mainly targets relational databases and suggests changing the physical storage when a new version of the schema needs to be materialized. In [35] the authors introduced the Bi-directional Database Evolution Language BiDEL as a solution to automatically generate queries that match the different structures within a relational database. Therefore, the users formulate their queries regardless of the schema version. This solution does not address record overlapping and is designed to support schema versioning in relational databases (where the schema should be defined before loading the data), whereas NoSQL databases store the data without any prior data validation or structure verification. Recent work considers schema versioning in the context of NoSQL stores. In [36], the authors introduce forward and backward query rewriting for querying data with different versions. Furthermore, it is possible to have heterogeneity within the same database, e.g., different cardinally, and different structures. However, the approach is limited to solving heterogeneity within a single collection, it requires a history graph of schema evolutions to enable query rewriting (which is not necessarily available), and it does not address record overlapping.

Schema-independent querying. This class of work proposes solutions to overcome schema heterogeneity by enabling schema-independent querying; in particular, the common strategy is to rely on query rewriting techniques [37] to reformulate an input query into several derivations, thus overcoming schema heterogeneity. Most research work is designed in the context of relational databases, where heterogeneity is usually restricted to the lexical level. When it comes to the hierarchical nature of semi-structured data (XML, JSON documents), the problem of identifying similar nodes is insufficient to resolve the problem of querying documents with structural heterogeneity for instance. To this end, keyword querying has been adopted in the context of XML [38]. The process of answering a keyword query on XML data starts with the identification of the existence of the keywords within the documents without the need to know the underlying schemas. The problem is that the results do not consider heterogeneity in terms of nodes, but assume that if the keyword is found, no matter what its containing node is, the document is adequate and must be returned to the user. Recent research work introduced a transparent querying mechanism to enable querying for heterogeneous documents. In [39], the authors introduced novel querying mechanisms based on query rewriting techniques [36] where they overcome the problem of structural heterogeneity in document stores. Their contribution consists of generating a dictionary with different attributes and their corresponding paths. Then, a query reformulation engine enriches the initial user queries with all possible paths extracted from the dictionary. In the same direction, another research work [40] resolves the problem of querying heterogeneous documents by covering a broader class of heterogeneity. Thus, the authors resolve the problem of having heterogeneous attributes that are semantically equivalent but with a *different naming convention*, as highlighted in [41], using a set of schema mappings. However, the queries must be combined to retrieve data from different structures. Overall, we notice that most of the schema-independent querying approaches consider the heterogeneity problem inside one collection at a time for a particular data model only. Furthermore, the resolution of schema heterogeneity is usually limited to a given type of heterogeneity. For instance, structural, or semantic whereas more classes of heterogeneity could be identified in NoSQL stores. For instance, the same information could be represented using *different data types*, and transcoding functions are required to resolve this heterogeneity [42].

Schema inference. To assist the users while formulating their queries, several research efforts have been directed toward schema inference techniques. The idea is to provide users with an overview of the different elements present in the heterogeneous data, e.g., document stores, [43,44]. This family of work was first introduced for inferring structures from semi-structured documents encoded in XML format. These papers aim to infer structures using regular expression rules from the different strings representing elements from XML documents to propose a generalized structure [45]. Both, JSON and XML are commonly used to encode nested data as documents. However, most of the solutions introduced to infer structures from documents encoded in XML could not be applied to documents encoded in JSON. Furthermore, other efforts were conducted to infer RDF data [46]. The problem with this class of work is none of these approaches is designed to deal with massive datasets whereas current applications are data intensive, and they are using JSON encoding.

In [41], the authors propose a framework to efficiently discover the existence of fields or sub-schemas inside the collection. To this end, the framework is built for managing a schema repository for JSON document stores. The proposed approach relies on a notion of JSON schema called skeleton, i.e., a tree representation describing the structures that frequently appear in a collection of heterogeneous documents. Thus, the skeleton may lack some paths that do exist in some of the documents because they do not appear often, and the generation of the skeleton will exclude them. Similarly, in [47] is proposed a schema profiling approach for document stores, where the goal is to expose the rules that drive the usage of different schemas to represent the same data. However, this approach is focused on providing insights into the users, but it does not provide any querying mechanism. In [48], a novel technique is defined to explain the schema variants within a collection in document stores. Therefore, the heterogeneity problem in this research work is detected when the same attribute is represented differently, e.g., different types, different locations inside documents. Therefore, the authors suggest using mappings to find out the different variations for a given attribute and ultimately build a multidimensional integrated view of the data to support OLAP queries. The main limitations of this approach are that it focuses on one collection at a time and that the query rewriting mechanism creates one query for every schema variation detected in such collection.

Overall, these works infer the implicit structures from heterogeneous data and provide the user with a high-level illustration regarding all or a subset of structures present inside the heterogeneous data. Schema inference techniques could help users to better understand the different underlying structures and to take the necessary measures and decisions during the application design phase. The limitation with such a logical view is that it requires a manual process to build the desired queries by including the desired attributes and all their possible navigational paths. In such approaches, the user is aware of data structures but is required to manage the heterogeneity. Furthermore, some proposals do not consider all structures and build an inferred schema on top of most used attributes, for instance, using some probability measures. Thus, queries could result in misleading results. Also, most of the proposals do not offer automatic support for structural evaluations and it is mandatory to regenerate the inference process which could affect the associated workloads and applications.

6.2 Data model heterogeneity

Multistore and polystores. In this part, we consider multistore and polystore systems providing integrated access and querying to several heterogeneous stores through a mediator layer. Systems like Teradata [49] or HadoopDB [50] propose to partition data between stores. Furthermore, they allow queries to access data shredded in the different stores and to move processing and/or data between stores. Such solutions require to co-locate stores within the same physical nodes to reduce the traffic overheads between nodes since data has to be moved to execute the queries, and different systems have to share each others' partitioning strategies. More recent proposals ensure access to the data using either a novel unified querying language (e.g., SQL++ [51]) or by supporting both the query languages of the underlying stores and a unified querying language (e.g., Spark SQL [52]). In [53] authors leverage several databases and processing platforms, and they define a unified declarative processing interface to access and query heterogeneous data. Another alternative to accessing the data is to formulate several queries using the different underlying stores querying languages and to employ a middle-ware layer to merge and return the final results [54]. More recent proposals consider wider support of integrated systems; for instance, ESTOCADA [55] supports key-value, document, relational, and nested relational data stores. Overall, we notice that despite the efficient support of different data models, the proposed multistore and polystore systems do not support schema heterogeneity.

Multi-model systems. In contrast to multistore systems (where data is stored in different stores), multi-model systems offer a single database to store and manage different data models by offering an integrated system to guarantee large scale databases requirements in terms of storage, availability, and fault tolerance (e.g., OrientDB, http://orientdb.com/orientdb/). The concept of multi-model systems was earlier introduced in the literature with the ORDBMS systems (object-relational database management systems) offering support to object-oriented programming with relational databases [56]. Recent multi-model systems advocate the idea of reducing the task of combining partial results from different stores and thus suggest having an integrated database, which hides the heterogeneity in terms of data models by providing a declarative approach of querying multi-model data. Therefore, data model transformation can be carried out only when it is required. In [57], this philosophy is embraced to propose a multi-model approach to data warehousing.

Ultimately, multi-model systems excel in terms of data governance, management, and access. It is only required to maintain one system while taking advantage of several data models. However, existing systems are limited to a pre-defined set of data models, extending support to new data models is challenging, and (most importantly) they do not provide any mechanism to handle schema heterogeneity (e.g., reconciling the usage of different naming conventions for the same attribute) nor record overlapping.

6.3 Record overlapping

Effectively supporting querying on a heterogeneous system with overlapping records requires the adoption of data fusion techniques [58]. The literature on this subject is very wide, thus we refer the reader to a recent survey [59]. Among the most important ones, we outline [19], where the authors propose a relational algebra operator (called *full-outerjoin merge*) to carry out data fusion while joining two tables—which is also the inspiration for the definition of the merge operator introduced in Sect. 4.1. Remarkably, related works in this area do not apply directly to a polyglot system; their scope is focused on the recognition and resolution of conflicts between records representing the same entity, but their application is mostly independent of the contextual storage and querying system. The related literature dealing with this problem is a building block of our approach, but it is not sufficient to address our complex multistore scenario on its own.

To the best of our knowledge, the only proposal that considers a scenario requiring data fusion in a polyglot system is QUEPA [60], where the authors present a polystore-based approach to support query augmentation. The idea is to let the user issue a query onto a single DBMS (using its native query language) and to augment query results with related information taken from the other DBMSs. The approach is meant to complement the other polystore systems that actually support cross-DBMS querying, and record linkage techniques are only used to find related instances in different DBMSs, but not to solve conflicts. Unlike [60], we (i) offer an integrated dataspace view over the whole multistore, (ii) enable cross-DBMS querying, and (iii) apply data fusion techniques at query time to solve conflicts in the data and return a polished result.

7 Conclusions

Data Science and Business Intelligence 2.0 expect more lightweight and flexible approaches to data analysis. Our proposal extends previous multistore solutions by handling schema heterogeneity under record overlapping and ensuring consistent answers for GPSJ queries, i.e., a wide class of queries that is the most common in OLAP. We rely on a lightweight pay-as-you-go approach to build an integrated dataspace to be used as an interface for query formulation; the formalized algorithms describe the process to obtain an execution plan from a GPSJ query and include several optimizations. The experimental evaluation measures the performance of the approach in terms of efficiency and shows how the pay-as-you-go approach can increasingly improve the effectiveness of query answering.

We plan to continue our research in several directions. Currently, the goal of our approach is to define an executable query plan that is semantically correct and that complies with the GPSJ semantics; whereas we do adopt some techniques to obtain a reasonably optimized query plan, we are not guaranteed that the best plan is identified. To this end, (1) a cost model would be necessary to estimate the cost of different plans and choose the best one, and (2) we plan to further increase the complexity of the algorithms to consider additional rationales to build execution plans. In particular, the rationale in the current implementation is to first merge the records at the entity level and then merge the reconciled entities—which guarantees the correctness of the result. A different approach would be to first compute local results at the database level and then merge them at the middleware level to obtain the global result. While this may seem a simple problem of reordering collection plans, it entails a correctness problem due to the presence of overlapping records: indeed, merging local results from different databases may require merging data belonging to different entities at the same time—which is not straightforward. Thus, we plan to investigate this issue to generate execution plans that can exploit database locality without compromising the correctness of query results. Further research efforts include adding support to the graph data model, enabling a broader set of queries than GPSJs (e.g., [10]), introducing KPIs to provide further insights into the user concerning the underlying heterogeneity of the data (e.g., [48]), and improving the efficiency of the schema extraction task by adopting approximation techniques (e.g., sampling to avoid a full scan of every collection) and an incremental strategy (i.e., to consider only new/updated records).

References

1. Sadalage, P.J., Fowler, M.: NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Pearson Education (2013)
2. 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)
3. DENODO corporation. https://www.denodo.com/. Accessed: 2021-02-02
4. Tan, R., Chirkova, R., Gadepally, V., Mattson, T.G.: Enabling query processing across heterogeneous data models: A survey. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 3211–3220. IEEE (2017)
5. Ben Hamadou, H., Gallinucci, E., Golfarelli, M.: Answering GPSJ queries in a polystore: A dataspace-based approach. In: Conceptual Modeling - 38th International Conference, ER 2019, Salvador, Brazil, November 4-7, 2019, Proceedings, vol. 11788, pp. 189–203. Springer (2019)
6. Franklin, M.J., Halevy, A.Y., Maier, D.: From databases to dataspaces: a new abstraction for information management. SIGMOD Record **34**(4), 27–33 (2005)
7. Gupta, A., Harinarayan, V., Quass, D.: Aggregate-query processing in data warehousing environments. In: 21th Int. Conf. on Very Large Data Bases, pp. 358–369. Morgan Kaufmann (1995)
8. Thomas, S.J., Fischer, P.C.: Nested relational structures. Adv. Comput. Res. **3**, 269–307 (1986)
9. Botoeva, E., Calvanese, D., Cogrel, B., Xiao, G.: Expressivity and complexity of mongodb queries. In: 21st Int. Conf. on Database Theory, pp. 9:1–9:23. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)
10. Ben Hamadou, H., et al.: Schema-independent querying for heterogeneous collections in NoSQL document stores. Inf, Syst (2019). In press
11. Golfarelli, M., Rizzi, S.: Data warehouse design: Modern principles and methodologies. McGraw-Hill, Inc. (2009)
12. Mazumdar, S., Seybold, D., Kritikos, K., Verginadis, Y.: A survey on data storage and placement methodologies for cloud-big data ecosystem. J. Big Data **6**(1), 15 (2019)
13. Rafique, A., Van Landuyt, D., Reniers, V., Joosen, W.: Towards an adaptive middleware for efficient multi-cloud data storage. In: Proceedings of the 4th Workshop on CrossCloud Infrastructures & Platforms, pp. 1–6 (2017)
14. National Center for Health Statistics: International classification of diseases, ninth revision, clinical modification (ICD-9-CM). https://www.cdc.gov/nchs/icd/icd9cm.htm. Accessed: 2021-02-02
15. Zhang, C., Lu, J., Xu, P., Chen, Y.: Unibench: A benchmark for multi-model database management systems. In: R. Nambiar, M. Poess (eds.) Performance Evaluation and Benchmarking for the Era of Artificial Intelligence - 10th TPC Technology Conference, TPCTC 2018, Rio de Janeiro, Brazil, August 27-31, 2018, Revised Selected Papers, vol. 11135, pp. 7–23. Springer (2018)
16. Bleiholder, J., Naumann, F.: Declarative data fusion - syntax, semantics, and implementation. In: J. Eder, H. Haav, A. Kalja, J. Penjam (eds.) Advances in Databases and Information Systems, 9th East European Conference, ADBIS 2005, Tallinn, Estonia, September 12-15, 2005, Proceedings, vol. 3631, pp. 58–73. Springer (2005)
17. Bernstein, P.A., Madhavan, J., Rahm, E.: Generic schema matching, ten years later. Proc. VLDB Endowment **4**(11), 695–701 (2011)
18. Maßmann, S., Raunich, S., Aumüller, D., Arnold, P., Rahm, E.: Evolution of the COMA match system. In: Proceedings of the 6th International Workshop on Ontology Matching, Bonn, Germany, October 24, 2011 (2011)
19. Naumann, F., Freytag, J.C., Leser, U.: Completeness of integrated information sources. Inf. Syst. **29**(7), 583–615 (2004)
20. Greco, S., Pontieri, L., Zumpano, E.: Integrating and managing conflicting data. In: D. Bjørner, M. Broy, A.V. Zamulin (eds.) Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia, July 2-6, 2001, Revised Papers, vol. 2244, pp. 349–362. Springer (2001)
21. Steinbrunn, M., Moerkotte, G., Kemper, A.: Heuristic and randomized optimization for the join ordering problem. VLDB J. **6**(3), 191–208 (1997)
22. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume I, *Principles of computer science series*, vol. 14. Computer Science Press (1988)
23. Frozza, A.A., dos Santos Mello, R., de Souza da Costa, F.: An approach for schema extraction of JSON and extended JSON document collections. In: 2018 IEEE International Conference on Information Reuse and Integration, IRI 2018, Salt Lake City, UT, USA, July 6-9, 2018, pp. 356–363. IEEE (2018)
24. Klettke, M., Störl, U., Scherzinger, S.: Schema extraction and structural outlier detection for json-based nosql data stores. In: Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs “Datenbanken und Informationssysteme” (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings, pp. 425–444. GI (2015)
25. Tahara, D., Diamond, T., Abadi, D.J.: Sinew: a SQL system for multi-structured data. In: 2014 ACM SIGMOD Int. Conf. on Management of Data, pp. 815–826. ACM (2014)
26. DiScala, M., Abadi, D.J.: Automatic generation of normalized relational schemas from nested key-value data. In: 2016 ACM SIGMOD Int. Conf. on Management of Data, pp. 295–310. ACM (2016)
27. Yeung, G.C., Gruver, W.A.: Multiagent immediate incremental view maintenance for data warehouses. IEEE Trans. Syst., Man, Cybernet.-Part A: Syst. Humans **35**(2), 305–310 (2005)
28. Sheth, A.P.: Federated database systems for managing distributed, heterogeneous, and autonomous databases. In: 17th Int. Conf. on Very Large Data Bases, p. 489. Morgan Kaufmann (1991)
29. Han, J., Haihong, E., Le, G., Du, J.: Survey on nosql database. In: 2011 6th international conference on pervasive computing and applications, pp. 363–366. IEEE (2011)
30. Amer-Yahia, S., Du, F., Freire, J.: A comprehensive solution to the xml-to-relational mapping problem. In: Proceedings of the 6th annual ACM international workshop on Web information and data management, pp. 31–38. ACM (2004)
31. Böhme, T., Rahm, E.: Supporting efficient streaming and insertion of xml data in rdbms. In: DIWeb, pp. 70–81 (2004)
32. Florescu, D., Kossmann, D.: Storing and querying xml data using an rdmbs. IEEE Data Eng. Bull. **22**, 3 (1999)
33. Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, K.S., Kersten, M.L.: Monetdb: Two decades of research in column-oriented database architectures. IEEE Data Eng. Bull. **35**(1), 40–45 (2012)
34. Hillenbrand, A., Levchenko, M., Störl, U., Scherzinger, S., Klettke, M.: Migcast: putting a price tag on data model evolution in nosql data stores. In: Proceedings of the 2019 International Conference on Management of Data, pp. 1925–1928 (2019)
35. Herrmann, K., Voigt, H., Behrend, A., Rausch, J., Lehner, W.: Living in parallel realities: Co-existing schema versions with a bidirectional database evolution language. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1101–1116 (2017)
36. Möller, M.L., Klettke, M., Hillenbrand, A., Störl, U.: Query rewriting for continuously evolving nosql databases. In: International Conference on Conceptual Modeling, pp. 213–221. Springer (2019)

37. Papakonstantinou, Y., Vassalos, V.: Query rewriting for semistructured data. In: ACM SIGMOD Record, vol. 28, pp. 455–466. ACM (1999)
38. Lin, C., Wang, J., Rong, C.: Towards heterogeneous keyword search. In: Proceedings of the ACM Turing 50th Celebration Conference-China, p. 46. ACM (2017)
39. Ben Hamadou, H., Ghozzi, F., Péninou, A., Teste, O.: Towards schema-independent querying on document data stores. In: 20th Int. Workshop on Design, Optimization, Languages and Analytical Processing of Big Data co-located with EDBT/ICDT. CEUR-WS.org (2018)
40. Gallinucci, E., Golfarelli, M., Rizzi, S.: Approximate OLAP of document-oriented databases: A variety-aware approach. Inf. Syst. **85**, 114–130 (2019)
41. Wang, L., Zhang, S., Shi, J., Jiao, L., Hassanzadeh, O., Zou, J., Wangz, C.: Schema management for document stores. Proc. VLDB Endowment **8**(9), 922–933 (2015)
42. Golfarelli, M., et al.: OLAP query reformulation in peer-to-peer data warehousing. Inf. Syst. **37**(5), 393–411 (2012)
43. Baazizi, M.A., Lahmar, H.B., Colazzo, D., Ghelli, G., Sartiani, C.: Schema inference for massive json datasets. In: (EDBT), pp. 222–233 (2017)
44. Ruiz, D.S., Morales, S.F., Molina, J.G.: Inferring versioned schemas from NoSQL databases and its applications. In: Proc. ER, pp. 467–480 (2015)
45. Freydenberger, D.D., Kötzing, T.: Fast learning of restricted regular expressions and dtds. Theor. Comput. Syst. **57**(4), 1114–1158 (2015)
46. Čebirić, Š., Goasdoué, F., Manolescu, I.: Query-oriented summarization of rdf graphs. Proceedings of the VLDB Endowment **8**(12), 2012–2015 (2015)
47. Gallinucci, E., Golfarelli, M., Rizzi, S.: Schema profiling of document-oriented databases. Inf. Syst. **75**, 13–25 (2018)
48. Gallinucci, E., Golfarelli, M., Rizzi, S.: Approximate OLAP of document-oriented databases: A variety-aware approach. Inf, Syst (2019). In press
49. Xu, Y., Kostamaa, P., Gao, L.: Integrating hadoop and parallel dbms. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 969–974 (2010)
50. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A.: Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. Proceedings of the VLDB Endowment **2**(1), 922–933 (2009)
51. Ong, K.W., Papakonstantinou, Y., Vernoux, R.: The sql++ query language: Configurable, unifying and semi-structured. arXiv preprint arXiv:1405.3631 (2014)
52. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu: Spark sql: Relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD, pp. 1383–1394. ACM (2015)
53. Lim, H., Han, Y., Babu, S.: How to fit when no one size fits. In: CIDR, vol. 4, p. 35. Citeseer (2013)
54. Gadepally, V., et al.: The bigdawg polystore system and architecture. In: 2016 IEEE High Performance Extreme Computing Conf., pp. 1–6. IEEE (2016)
55. Bugiotti, F., et al.: Invisible glue: Scalable self-tunning multi-stores. In: 7th Biennial Conf. on Innovative Data Systems Research. www.cidrdb.org (2015)
56. Hall, B., Lunetta, M.: Object relational database management system (2003). US Patent App. 10/122,088
57. Bimonte, S., Gallinucci, E., Marcel, P., Rizzi, S.: Data variety, come as you are in multi-model data warehouses. Information Systems p. 101734 (2021)
58. Bleiholder, J., Naumann, F.: Data fusion. ACM computing surveys (CSUR) **41**(1), 1–41 (2009)
59. Mandreoli, F., Montangero, M.: Dealing with data heterogeneity in a data fusion perspective: Models, methodologies, and algorithms. In: Data Handling in Science and Technology, vol. 31, pp. 235–270. Elsevier (2019)
60. Maccioni, A., Torlone, R.: Augmented access for querying and exploring a polystore. In: 34th IEEE Int. Conf. on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018, pp. 77–88. IEEE Computer Society (2018)