Information Systems Frontiers
https://doi.org/10.1007/s10796-022-10320-2

Cost-based Optimization of Multistore Query Plans

Chiara Forresi[1] · **Matteo Francia**[1] · **Enrico Gallinucci**[1] · **Matteo Golfarelli**[1]

Accepted: 31 July 2022

Abstract
Multistores are data management systems that enable query processing across different and heterogeneous databases; besides the distribution of data, complexity factors like schema heterogeneity and data replication must be resolved through integration and data fusion activities. Our multistore solution relies on a dataspace to provide the user with an integrated view of the available data and enables the formulation and execution of GPSJ queries. In this paper, we propose a technique to optimize the execution of GPSJ queries by formulating and evaluating different execution plans on the multistore. In particular, we outline different strategies to carry out joins and data fusion by relying on different schema representations; then, a self-learning black-box cost model is used to estimate execution times and select the most efficient plan. The experiments assess the effectiveness of the cost model in choosing the best execution plan for the given queries and exploit multiple multistore benchmarks to investigate the factors that influence the performance of different plans.

Keywords Multistore · NoSQL · Query optimization · Cost model

1 Introduction

The decline of the *one-size-fits-all* paradigm has pushed researchers and practitioners towards the idea of *polyglot persistence* (Sadalage & Fowler, 2013), where a multitude of databases is employed to support data storage and querying. The motivations are manifold, including the exploitation of the strongest features of each system, the off-loading of historical data to cheaper database systems, and the adoption of different storage solutions by different branches of the same company. This trend has also influenced the discipline of data science, as analysts are being steered away from traditional data warehousing and towards a more flexible and lightweight approach to data analysis. Multistore contexts are characterized by 1) the replication of data across different storage systems (i.e., there is no sharp horizontal partitioning) with possibly conflicting records (e.g., the same customer with a different country of residence in different databases), and 2) a high level of schema heterogeneity: records of the same real-world entity may be represented with different structures, using different naming conventions for the same kind of data. The large volume and the frequent evolution of these data hinder the adoption of a traditional integration approach.

In recent work (Ben Hamadou et al., 2019; Forresi et al., 2021; Forresi et al., 2021) we have proposed a multistore solution that relies on a dataspace to provide the user with an integrated view of the data. A *dataspace* is a lightweight integration approach providing basic query expressiveness on a variety of data sources, bypassing the complexity of traditional integration approaches and possibly returning best-effort or approximate answers (Franklin et al., 2005). The dataspace is built in accordance with a *pay-as-you-go* philosophy, i.e., by applying simple matching rules to recognize relationships between data structures and by letting the users progressively refine the dataspace as new relationships are discovered (Jeffery et al., 2008). Users exploit the dataspace to formulate GPSJ (generalized projection, selection and join) queries, i.e., the most common class of queries in OLAP applications (Golfarelli

✉ Enrico Gallinucci
enrico.gallinucci@unibo.it

Chiara Forresi
chiara.forresi@unibo.it

Matteo Francia
m.francia@unibo.it

Matteo Golfarelli
matteo.golfarelli@unibo.it

[1] DISI - Department of Computer Science and Engineering, University of Bologna, Via dell'Università 50, Cesena, 47522, Italy

Published online: 04 October 2022

et al., 1998). Queries are translated into execution plans that consist of many local computations (carried out by the single databases) and a global computation (carried out by the middleware layer).

In this paper, we propose a technique to optimize the execution of GPSJ queries by finding the most efficient execution plan on the multistore. The main challenge lies in devising a cross-database execution plan that couples data fusion operations with the resolution of schema heterogeneity and efficiently provides a correct result; remarkably, data fusion consists in merging duplicated records that refer to the same real-world entity into a single representation by resolving possible conflicts (Mandreoli & Montangero, 2019). In particular, the paper provides the following contributions. (1) The extension of the multistore scenario presented by Forresi et al. (2021) to consider data that is replicated across different databases. (2) The introduction of an advanced query planner that exploits different characteristic of the multistore and its data. (3) The presentation of a multi-database and self-learning cost model to compare the complexity of execution plans and choose the most efficient one. The cost model keeps into account both the execution plan features and the database resources. (4) The evaluation of the proposed on multiple multistores with different characteristics based on a realistic real-world scenario. An exploratory study in this direction (only limited to a simple example) had been done by Forresi et al. (2021); in this paper, the work is completed with the full formalization of the approach, the introduction of a realistic real-world scenario, the detailed discussion of the generated query plans, the improvement of the cost model, and the substantial extension of the experimental evaluation.

The paper is structured as follows. Sections 2 gives an overview of the multistore scenario and Section 3 discusses related work. In Sections 4 and 5 we introduce the background knowledge on our multistore and the multistore algebra that is extended from Nested Relational Algebra (NRA), while Section 6 presents the approach to query planning. In Section 7 we discuss the cost model and Section 8 shows the experimental evaluation. The conclusions are drawn in Section 9.

2 Multistore Overview

The multistore operates under the following assumptions.

1. Being a multistore, there exist multiple databases, supporting a variety of data models: relational, document-based, wide-column, and key-value.
2. Records may be replicated in collections of different databases, possibly with conflicting values.

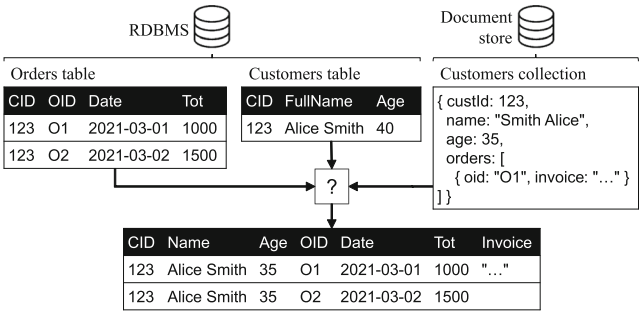

Fig. 1 An exemplification of data model heterogeneity, schema heterogeneity, and record overlapping in a multistore

3. Schema heterogeneity is present at different levels: records representing the same kind of entity (e.g., a customer) may have missing (or additional) attributes or adopt different naming conventions; this issue may occur between different collections, but also within the same collection (due to the schemaless nature of NoSQL databases).

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.

As discussed by Forresi et al. (2021), the above conditions are met in many real-world applications, which – in practical contexts – often refer to data virtualization systems for data analysis. Below we describe two specific contexts that emerged during our interaction with Denodo,[1] 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 (Golfarelli & Rizzi, 2009). 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,

[1] https://www.denodo.com

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 (Mazumdar et al. 2019). 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 (Rafique et al., 2017). 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,[2] 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 health unit can store such data in different data models and schemas, depending on the adopted software.

In this paper, we consider a *multi-cloud architecture* case study, where different branches of the same holding store the same data but rely on different storage systems. Figure 2 shows a conceptual view through an ER diagram. Figure 3 shows the physical implementation. C_1 to C_7 represent the collections of data, while the ":" notation is used to indicate the entities appearing 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 database, Cloud 2 satisfies the need for data variety support by relying on NoSQL systems; also notice that Cloud 2 additionally stores orders' invoices. As the two branches belong to the same holding, both customers

[2]International Classification of Diseases, https://www.cdc.gov/nchs/icd/icd9cm.htm

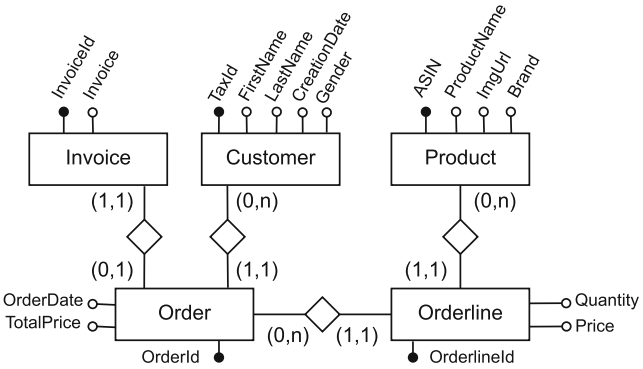

Fig. 2 The ER diagram of the case study

and products are partially overlapped in the two cloud environments.

Figure 4 provides a functional overview of the multistore system and the supported user interactions. For starters, the user interacts with the system to create and continuously refine the dataspace, i.e., an abstract global representation of the data in the multistore. The dataspace is composed of two main concepts: *entities*, corresponding to the real-world entities in the multistore (e.g., customers, products), and *features*, corresponding to unique representations of the attributes describing entities (e.g., the name of customers, the brand of products). As discussed by Forresi et al. (2021), the dataspace is built and maintained through an incremental and semi-automatic approach that embraces the pay-as-you-go philosophy (Jeffery et al., 2008).

The dataspace is used to formulate GPSJ queries, which are well-suited for data analysis; a typical OLAP query consists of a group-by set (i.e., the features used to carry out an aggregation), one or more numerical features to be aggregated by some function (e.g., sum, average), and (possibly) some selection predicates. A GPSJ queries formulated on the case study would aggregate events (e.g., Orderline and Order) to calculate KPIs or measures (e.g.,

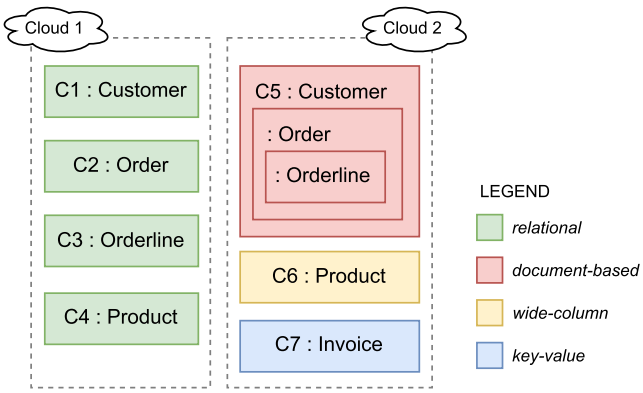

Fig. 3 A graphical representation of the physical implementation of the case study. Different colors represent different databases with different data models

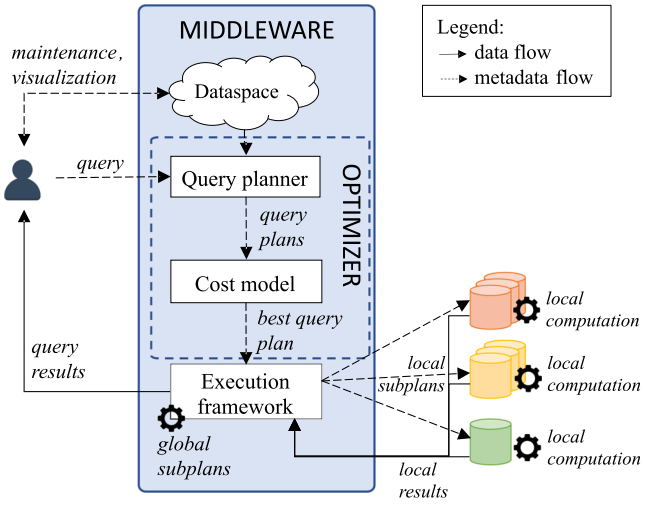

Fig. 4 Overview of our multistore

the total Quantity, the average Price) by a grouping criteria (e.g., by Product, by Customer). Based on the user's query, the system's Optimizer defines the query plan to be executed on the multistore in two steps: first, the Query planner generates multiple query plans, then a Cost model is used to choose the most convenient one. Query plans are decomposed into subplans, each identified by *macro operators* that embed a tree of operations. *Local subplans* are computed directly on the local databases; *global subplans* are computed on the middleware's execution framework to combine the partial results from local subplans and obtain the final result to return to the user.

3 Related Work

3.1 Multistore Systems

The variety in terms of data models 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 (Tan et al. 2017). A naive approach to solve the problem of querying several data models is to transform all datasets into a reference data model — usually the relational one (DiScala & Abadi, 2016). This kind of solution leads to the loss of the schemaless flexibility and requires continuous maintenance to support schema evolution. A different approach is proposed by *multimodel* systems, which directly support several data models within the same platform; examples of multimodel databases are OrientDB[3] and ArangoDB.[4] Inter-data model querying is enabled by a custom query language to support nested structures and graph queries. Besides being limited to a predefined set of data models, these systems do not directly support data fusion. Indeed, a data warehousing approach relying on a multimodel system is presented by Bimonte et al. (2021), but neither schema heterogeneity nor data fusion is tackled.

In recent years, *multistore* and *polystore* have emerged to provide integrated access and querying to several heterogeneous stores through a mediator layer (middleware) (Tan et al. 2017). The difference between multistores and polystores lies in whether they offer a single or multiple querying interfaces, respectively. Among the most notable are BIGDAWG (Gadepally et al., 2016), TATOOINE (Bonaque et al. 2016), and CloudMDsQL (Kolev et al. 2016). These systems vary in the functionalities they support (e.g., the available data models and storage systems, the support to the ingestion process, the expressiveness of the querying language, the possibility to move data from one database to another). However, none of them supports data fusion. Remarkably, adopting these approaches in a scenario with duplicated records would require to carry out a pre-processing activity to clean duplicates and resolve conflicts, which may be hindered (i) by the inability to overwrite data in the original databases and/or (ii) by the frequent evolution of data and schemas. Otherwise, failing to deal with duplicated records may lead to wrong results and, ultimately, to inaccurate decision-making.

Effectively supporting querying on a heterogeneous system with overlapping records requires the adoption of data fusion techniques (Bleiholder & Naumann, 2008). The literature on this subject is very wide, thus we refer the reader to a recent survey (Mandreoli & Montangero, 2019). Remarkably, related work in this area does not apply directly to a polyglot system. To the best of our knowledge, the only proposal that considers a scenario requiring data fusion in a polyglot system is QUEPA (Maccioni & Torlone, 2018), where the authors present a polystore-based approach to support query augmentation. The approach is meant to complement the other polystore systems that actually support cross-database querying, and record linkage techniques are only used to find related instances in different databases, but not to solve conflicts. Another work that proposes on-the-fly integration and schema heterogeneity resolution in an analytical context is (Gallinucci et al., 2019); however, the proposed approach is limited to document-oriented databases and does not consider data fusion.

3.2 Multistore Optimization

Besides the support to querying over heterogeneous databases, multistore systems must adopt optimization

[3] OrientDB, https://orientdb.org/

[4] ArangoDB, https://www.arangodb.com/

strategies to define efficient execution plans. In the literature, these optimization strategies are usually implemented through rule-based mechanisms and cost models; the main differences lie in the abstraction of the execution plan (i.e., whether it is considered as a whole or it is subdivided into its local and global parts) and the level at which they operate (i.e., at the logical or physical level).

Rule-based optimization consists in defining a set of rules to reorder the operations within logical plans (e.g., by bringing selections closer to the start of the plan) (Gog et al., 2015) or to choose the most efficient algorithm in the translation of a logical plan to a physical one (Wang et al., 2017). Solely relying on rule-based optimization is considered impractical and ineffective: these rules typically make very simplistic decisions based on the different cardinality and complexity of each operator (Agrawal et al., 2018), whereas the cost actually depends on many input parameters (e.g., selectivity, disk I/O, CPU cycles). Moreover, as new platforms and applications emerge, maintaining a rule-based optimizer becomes cumbersome as the number of rules grows rapidly.

Cost-based optimization relies on cost models to estimate the cost of query plans (usually in terms of time or I/O operations) and to choose the one that minimizes such cost. Some approaches are limited to the optimization of local databases computations — using either a single (Agrawal et al., 2018) or multiple cost models (Deshpande & Hellerstein, 2002) —, while others adopt a global optimization technique to decide which computations should be pushed down to local databases (Sellami & Defude, 2018). In general, using distinct cost models for each engine increases the accuracy of the optimization and enables a finer characterization (Duggan et al., 2015) (e.g., different databases may be more or less efficient in carrying out the same operations (Forresi et al., 2021)). The accuracy of cost models usually depends on the level of detail that they capture; however, the risk in fine-grained cost modeling is to make the training phase too complex and expensive (Singhal et al., 2019). To this end, an *active learning* technique (Golfarelli et al., 2019) can be used, as it allows for a simpler initial training and continuously improves the accuracy of the model by automatically updating it as queries are issued by the users. In the literature, cost models are often distinguished between white-box and black-box models.

- *White-box* cost models associate theoretical formulas to the different query operators and build up the cost of a query by summing the cost of each operation. In multistore systems, white-box approaches usually require a separate cost model for each execution engine, as the latter are too different to be represented by a unique cost model (Bondiombouy & Valduriez, 2016). Some works break down the problem and focus only on either the local databases (Deshpande & Hellerstein, 2002) or the middleware (Subramanian & Subramanian, 1998). Complete works are presented by Sellami and Defude (2018) and Agrawal et al. (2018), where queries are broken down into sets of logical operators, the cost of which is determined in terms of disk I/O, CPU, and network; however, the full details of the adopted formulas are not disclosed. A similar approach was adopted by Forresi et al. (2021), where the cost of query plan operations is estimated in terms of disk pages read and written on each execution engine.
- *Black-box* cost models hide the behavior of an execution engine within a black-box, where the known information is mostly limited to the issued queries and the given response times. The estimates for query execution times are then obtained by comparing the characteristics of the current query with those of the previously executed ones. Singhal et al. (2019) suggests using black-box modeling in a multistore/polystore system; this kind of approach is also used in non-strictly related contexts, such as Kaitoua et al. (2019) (where the goal is to optimize data migration between different databases) and Golfarelli et al. (2019) (where the goal is to estimate the cost of queries run on external web services, whose internal characteristics are not known).

Both approaches are not exempt from risks, as devising an effective cost model requires a careful evaluation of cost factors. On the one hand, white-box approaches require a deep understanding of each database's internal mechanisms - as well as inter-database communication patterns - in order to algorithmically reproduce the same behavior. On the other hand, black-box approaches require the identification of all features (of queries and databases) that influence the cost of queries.

Among the systems mentioned in Section 3.1: BIGDAWG (Gadepally et al., 2016; Duggan et al., 2015) implements black-box models for each execution engine, even though their usage is limited to the optimization of query portions within the same engine; TATOOINE (Bonaque et al. 2016) makes no mention of cost optimization; and CloudMDsQL (Kolev et al. 2016) blends rule-based optimization with a combination of both white and black-box cost modeling, but discloses no implementative details. The multistore proposed in this paper relies on rule-based optimization for a logical reordering of operations at both local and global levels, and uses a black-box cost model with active learning to identify the most efficient execution plan. The choice of a black-box cost model allows to overcome the many challenges behind white-box ones, which either require an enormous effort to effectively model the many factors that contribute to query costs in a complex and

heterogeneous environment like a multistore, or suffer the limitations due to the assumptions that must be made to keep the model simple. Indeed, black-box cost models automatically learn and continuously fine-tune a model of the systems' behavior, thus unloading the burden of this task from the user.

4 Multistore Formalization

4.1 Basic Concepts

The multistore is described by a *dataspace*, i.e., an abstract global representation of the data scattered across different databases. It is composed of two main concepts: *entities*, corresponding to the real-world entities in the multistore (e.g., customers, products), and *features*, corresponding to the attributes that describe entities (e.g., the name of customers, the brand of products). These concepts are built in a pay-as-you-go fashion by analyzing the schemas in the data and detecting relationships between attributes.

The coexistence of different databases and the absence of a unique reference data model carries the need for common terminology. Consistently with the terminology proposed by Atzeni et al. (2014), we refer to *collections* as the data structures that contain sets of records associated with a schema, and to *records* as 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-based 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. We do not consider documents and rows as a whole, but we separately model the records available at each nesting level. Differently from Atzeni et al. (2014), we use the term *attribute* (instead of *field*) to refer to the single properties of each record.

Definition 1 (Collection, Record, Attribute) A collection C is a set of records; a record $r = \{(a_1, v_1), \ldots, (a_n, v_n)\}$ is a set of key-value pairs, where each value v_i is associated to an attribute a_i. Values are either of primitive type (e.g., number or string) or arrays of records.

From this point forward, we refer to *primitive attributes* or *array attributes* based on the type of associated values. If an attribute is nested within one or more array attributes, its name includes the dot-concatenation of the names of those array attributes.

Example 1 Figure 5 shows a sample document of a document-based database, corresponding to a customer, her orders, and the respective order details; the two order line records (in orange) are nested within the order record (in green), which is nested itself within the customer record (in blue).

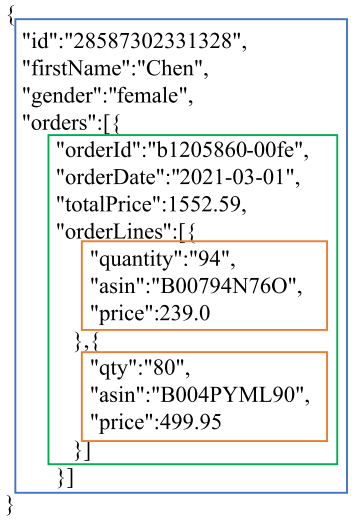

Fig. 5 A sample document corresponding to a customer, her orders, and the respective order details; four *records* are shown in the boxes, and each color (blue, green, and orange) corresponds to a different *schema*

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

Definition 2 (Schema) A schema $S = \{a_1, \ldots, a_{|S|}\}$ is a set of attributes that applies to one or more records in a collection C. Each schema has a key attribute $key(S)$, whose values uniquely identify the records with schema S within C.

A record r with schema S in a collection C may be nested within another record r'; in such a case, $arr(S, C)$ denotes the array attributes of r' that contain r. The set of all schemas in the dataspace is indicated with $\mathcal{S}$. For the sake of simplicity, we assume all keys to be simple. Given a record r, its schema (denoted with S_r) is the set of attributes directly available in r. If r is contained within an array attribute a of a record t, then (i) in S_r we also include $key(S_t)$ (this is necessary to maintain the relationship between the schema of a nested record and the one of the parent record[5]), and (ii) $arr(S_r, C)$ extends $arr(S_t, C)$ with a, so that $arr(S_r, C)$ provides the full

[5]The inclusion of $key(S_t)$ in S_r will allow a mapping to be defined between the attributes in S_r and S_t (see Definition 3), which – in turn – will enable the inference of entities and relationships (see Definition 8).

list of parent array attributes. 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).

Example 2 The sample document of a collection C shown in Fig. 5 contains four schemas:

- S_{bl} = {id, firstName, gender, orders}
- S_{gr} = {id, orders.orderId, orders.orderDate, orders.totalPrice, orders.orderLines}
- S_{or_1} = {orders.orderId, orders.orderLines.quantity, orders.orderLines.asin, orders.orderLines.price}
- S_{or_2} = {orders.orderId, orders.orderLines.qty, orders.orderLines.asin, orders.orderLines.price}

It is $arr(S_{bl}, C) = [\ ]$, $arr(S_{gr}, C) =$ [orders], and $arr(S_{or_1}, C) = arr(S_{or_2}, C) =$ [orders, orders.orderLines].

Due to both schema variability and schema denormalization, several attributes may be found in different schemas to 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.

Definition 3 (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 belonging to different schemas; $\varphi_{(a_i,a_j)}$ is a bijective transcoding function to express the values of a_j in the format of a_i (if no transcoding is necessary, $\varphi_{(a_i,a_j)} = I()$ where $I()$ is the identity function). The existence of a mapping between a_i and a_j is indicated with $a_i \equiv a_j$.

Mappings are considered between single attributes; given two schemas S_i, S_j and an attribute $a_i \in S_i$, we assume there exists at most an attribute $a_j \in S_j$ such that $a_i \equiv a_j$. Mappings recognize that there is a semantic equivalence between two attributes in different schemas, thus we introduce *features* to provide a unique reference for equivalent attributes in several different schemas.

Definition 4 (Feature) A *feature* represents a set of attributes mapped to each other. We define a feature f as a triple, $f = (a, M, \between)$, where:

- a is the *representative* attribute;
- M is the set of mappings that link all the feature's attributes to the representative a. The set of attributes represented by f is indicated with $attr(f)$, and $\forall m_j \in M$ it is $m = (a, a_j, \varphi_{(a,a_j)})$.
- $\between : (v_i, v_j) \rightarrow v_k$ is an associative and commutative function that resolves the possible conflicts between the values v_i and v_j of any two attributes a_x and a_y represented by f by returning a (possibly new) value v_k.

Given a record r, the conflict resolution function $\between$ can be applied to $r[a_i]$ and $r[a_j]$ if $\{a_i, a_j\} \subseteq attr(f)$; we refer the reader to Bleiholder and Naumann (2005) for different methods to define conflict resolution functions. Notice that an attribute is always represented by one and only one feature; thus, for any two features f_i and f_j, it is $attr(f_i) \cap attr(f_j) = \varnothing$ (otherwise, it would mean that $\exists\, a : feat(a) \subseteq \{f_i, f_j\}$, but $|feat(a)| = 1$ by definition). We use $feat(a)$ to refer to the feature of an attribute a, $rep(f)$ to refer to the representative attribute of f. Features are defined even for attributes that are not associated to any mapping; in such a case, it is $|\ attr(f)\ | = 1$ and $M = \varnothing$. A feature is a *key* feature if its attributes act as a key in at least one schema (and possibly play the role of foreign keys in other schemas).

Features describe properties of real-world concepts (e.g., customers, products) that we refer to as *entities*.

Definition 5 (Entity) An entity E is an abstract representation of a real-world entity. It is associated with a set of features $F_E \subseteq F$ and it is identified by a *key* feature $key(E) \in F_E$; $\mathcal{S}_E$ denotes the set of schemas such that $\forall\, S \in \mathcal{S}_E$ it is $feat(key(S)) = key(E)$.

Remarkably, key features may be associated with multiple entities, while non-key features are associated with a single entity.[6]

The schemaless nature of NoSQL databases allows different records in the same collection with different key features. We say that a collection is *well-formed* if it maintains internal consistency, i.e., if there exists a single key feature. From this point forward, we assume all collections to be well-formed.

[6] At a first glance, this may look like a limitation: for instance, both entities for products and customers (i.e., E_{pr} and E_{cu}) are associated to different "name" features (i.e., f_2 and f_{12} in Table 1). Although it could make sense to infer a relationship between the respective attributes and model them through a single feature, it would open to ambiguities when the user uses such feature for grouping or selection purposes (i.e., would the user be interested in the name of products or the name of customers?). By associating non-key features to a single entity, the two names remain separate and ambiguities are prevented.

Table 1 An 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

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

Definition 6 (Collection well-formedness) A collection C is *well-formed* if there exists a feature $f = key(E)$ such that $\forall\, r \in C$ it is $key(S_r) = f$.

Relationships between entities are identified as follows.

- A many-to-one relationship from E_i to E_j on a feature f is indicated with $E_i \xrightarrow{f} E_j$; it is inferred if $\exists f$:

$\forall\ S_i \in \mathcal{S}_{E_i}, S_j \in \mathcal{S}_{E_j}$ it is $attr(f) \cap \{S_i \setminus key(S_i)\} \neq \emptyset \wedge key(S_j) \in attr(f)$. In other words, the many-to-one relationship is inferred if the attributes of f in the schemas of E_i are non-key attributes mapped to the keys of the schemas of E_j.

- A one-to-one relationship between E_i and E_j on f is indicated with $E_i \overset{f}{\leftrightarrow} E_j$; it is inferred if $\exists f : \forall\ S_i \in \mathcal{S}_{E_i}, S_j \in \mathcal{S}_{E_j}$ it is $key(S_i) \in attr(f) \wedge key(S_j) \in attr(f)$.

In a many-to-one relationship $E_i \overset{f}{\rightarrow} E_j$, E_j is *coarser* than E_i and E_i is *finer* than E_j. It is $E_i \Rightarrow E_k$ if there exists a path of -to-one relationships from E_i to E_k.

Example 3 Table 1 presents our motivating example 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 $a_{15} \equiv a_{16}$ since $feat(a_{15}) = feat(a_{16}) = f_6$. Mappings reveal the relationship between the schemas. It is $\mathcal{S}_{E_{pr}} = \{S_1, S_2, S_3\}$ since $key(S_1) \equiv key(S_2) \equiv key(S_3)$; similarly, it is $\mathcal{S}_{E_{ol}} = \{S_4, S_5\}$. From f_1 it is inferable a many-to-one relationship $E_{ol} \overset{f_1}{\rightarrow} E_{pr}$. 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 features (e.g., f_1) that overlap with more entities.

As discussed by Forresi et al. (2021), features and entities are obtained incrementally in a pay-as-you-go fashion (Jeffery et al., 2008). An automatic procedure is first run to discover mappings and infer the relationships to define features and entities; then, users can refine mappings and update the knowledge on features and entities, as long as the specified constraints are not violated.

4.2 Dataspace and Supporting Structures

The dataspace is the data structure that provides the integrated, high-level view of the data in the multistore.

Definition 7 (Dataspace) The dataspace $\mathcal{D}$ is a directed graph $\mathcal{D} = (\mathcal{E}, L)$ where $\mathcal{E}$ is the set of entities in the dataspace and L is the set of links (i.e., many-to-one or one-to-one relationships) between the entities. A *fact* of the dataspace is an entity E^* for which there is no finer entity in $\mathcal{D}$, i.e., $\nexists\ E \in \mathcal{E} : E \rightarrow E^*$; the set of facts in $\mathcal{D}$ is $fact(\mathcal{D}) = \mathcal{E}^* \subseteq \mathcal{E}$.

Example 4 Figure 6 shows the dataspace of the case study. Since customer and product records are overlapped, it is

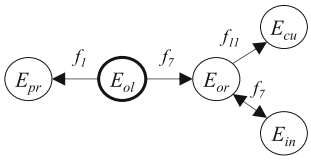

Fig. 6 The dataspace $\mathcal{D}$ of the case study

$\phi(E_{pr}) = \phi(E_{cu}) = true$, while $\phi(E_{ol}) = \phi(E_{or}) = \phi(E_{in}) = false$. The only fact entity is E_{ol} (shown with a bolder border).

To denote the relationships between collections and entities, we introduce the concept of *granularity* and the expressions of *entity representation* and *entity description* are used.

Definition 8 (Granularity, entity representation, and entity description) Let C a well-formed collection whose key feature f; its granularity $gran(C)$ is the entity E that corresponds to f, i.e., $key(E) = f$. If $gran(C) = E$, then we say that C *is represented by* E. C *is described by* E if there exists an attribute $a \in C$ such that $feat(a) \in F_E$, $feat(a) \neq key(E)$. A collection can be described by more entities, and an entity can describe more collections. The entity representing a collection (i.e., its granularity) also describes it.

A collection contains a portion of the multistore data that refers to some of the dataspace's entities. A *collection graph* is the portion of the dataspace indicating the entities describing a collection.

Definition 9 (Collection graph) Given a collection C, its collection graph $CG_C = (\mathcal{E}_C, L_C)$ is a subgraph of $\mathcal{D}$ limited to the entities of $\mathcal{D}$ that describe C.

When a collection is described by a single entity, its records contain only primitive attributes (i.e., first normal form) and values of all non-key attributes only depend on the key attribute (i.e., third normal form). In presence of two (or more) entities, the collection makes use of either *nesting* or *flattening*: we refer to nesting when non-primitive attributes are used and to flattening when values of non-key attributes depend on other non-key attributes.

Example 5 Figure 7 shows examples of different collection graphs representing the same set of entities, where E_{ol} is the only fact entity for each collection. C_i is a fully nested collection, where the granularity is E_{cu}; C_k is a fully flat collection, where the granularity is E_{ol} (indeed, there are two records, one for each order line, and order and customer attributes are duplicated); C_j is an intermediate

solution, where the granularity is E_{or}: customer attributes are flattened, whereas order line attributes are nested.

Collection graphs may be of arbitrary complexity. We distinguish three patterns of collection graphs, which are used by the multistore algebra. The key aspect in common is the presence of a single fact entity. Let $CG_C = (\mathcal{E}_C, L_C)$.

- A *normal graph* (NoR) is composed by a single entity. In this case, it is $L_C = \varnothing$ and $\mathcal{E}_C = E = gran(C) = fact(CG_C)$.
- A *nested graph* (NeR) is composed by at least two entities and there is a single path of edges that are directed towards $gran(C)$, i.e., there is only one fact entity $fact(CG_C)$, which is the one entity farthest away from $gran(C)$. In this case, $\forall\, E \in \{\mathcal{E}_C \setminus gran(C)\}$ it is $E \Rightarrow gran(C)$.
- A *flat graph* (FlR) is composed by at least two entities and all edges are directed away from $gran(C) = fact(CG_C)$. In this case, $\forall\, E \in \{\mathcal{E}_C \setminus gran(C)\}$ it is $gran(C) \Rightarrow E$.

If a collection graph conforms to one of these patterns, we indicate it with $rep(CG_C) \in \{\text{NoR}, \text{NeR}, \text{FlR}\}$. Notice that, given our internal consistency assumption on the granularity of collections, each collection is associated to a single collection graph.

Example 6 In Fig. 7, it is $rep(CG_{C_i}) = \text{NeR}$ and $rep(CG_{C_k}) = \text{FlR}$, while $rep(CG_{C_j}) = \varnothing$.

5 Multistore Algebra

In this section, we present the algebra we adopt to formulate query plans. Section 5.1 presents an extension of traditional Nested Relational Algebra (NRA) with data fusion operations; Section 5.2 introduces *entity views* as a high-level abstraction for query plans and the high-level operators to formulate query plans with entity views.

5.1 NRA and Data Fusion Operations

Query execution plans are formulated in Nested Relational Algebra (NRA). A summary of the supported NRA operators is shown in Table 2. In this paper, NRA is slightly extended to support the data fusion operations required by our multistore scenario.

The most important addition to NRA is the extension of the join operator's semantics to handle data fusion. We do this by relying on the *merge* operator ($\sqcup$) (Forresi et al., 2021), which addresses the extensional and intensional overlap between collections. In particular, given two collections represented by the same entity (e.g., customers), the merge operator exploits mappings to resolve schema heterogeneity (e.g., to recognize that the customers' names are reported in both collections with different naming conventions) and record overlapping (e.g., to produce a single record for customers that are replicated in both collections). Its goal is to keep as much information as possible, 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.

First, let us formalize the merge operation between two generic recordsets.

Definition 10 (Merge operation) Let R_i and R_j be the recordsets of two schemas S_i and S_j, with $a_k \in S_i$ and $a_l \in S_j$ such that $a_k \equiv a_l$, i.e., $\exists f : \{a_k, a_l\} \subseteq attr(f)$.

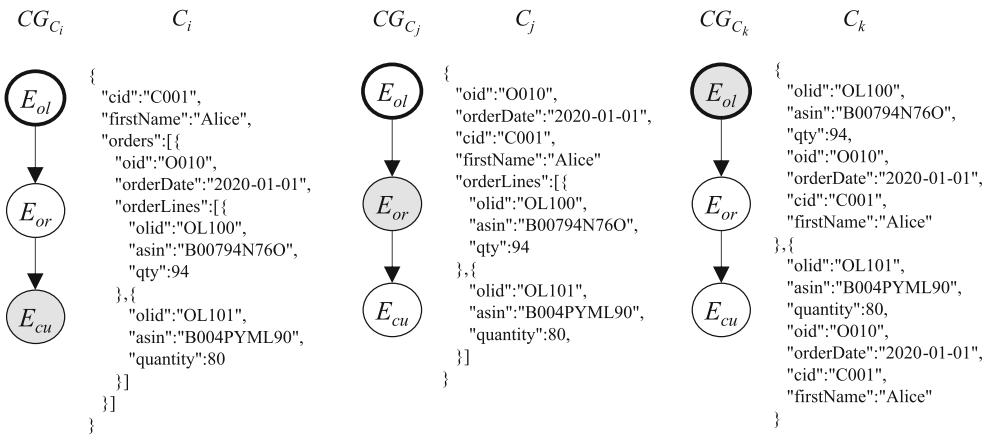

Fig. 7 Three examples of collection graphs representing the same data in different ways (i.e., $CG_{C_i}, CG_{C_j}, CG_{C_k}$), alongside the corresponding records in a document-based representation (i.e., C_i, C_j, C_k). The fact entity is in bold, while the granularity is in grey

Table 2 NRA operators

Operator	Description
$\pi_Y(C)$	Denotes a **projection** operation on collection C, where Y is a set of projection predicates.
$\sigma_x(C)$	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.
$\gamma_{(F,Z)}(C)$	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.
$\nu_{(F',a,Z)}(C)$	Denotes a **nest** operation on collection C, where F' is the group-by set (i.e., a set of features), a is an array attribute to be created, and Z is the set of features to be nested within a.
$\mu_a(C)$	Denotes the **unnest** of an array attribute a on collection C.
$(C_1) \bowtie_{(a_i,a_j)} (C_2)$	Denotes a **join** operation between collections C_1 and C_2 based on the join predicate $a_i = a_j$, with $a_i \in C_1$, $a_j \in C_2$. If $\exists f : \{a_k, a_l\} \subseteq attr(f)$, it can be shortened as $(C_1) \bowtie_f (C_2)$.
$(C_1) \sqcup_{(a_i,a_j)} (C_2)$	Denotes a **merge** operation between collections C_1 and C_2 based on the join predicate $a_i = a_j$, with $a_i \in C_1$, $a_j \in C_2$. If $\exists f : \{a_k, a_l\} \subseteq attr(f)$, it can be shortened as $(C_1) \sqcup_f (C_2)$. See Definition 10.

The merge of the two recordsets $R_i \sqcup_f R_j$ produces a recordset R_{ij} with schema $S_{ij} = S_i^* \cup S_j^* \cup S_{ij}^\cap$ such that:

- $S_i^* = \{a \in S_i : \nexists\, a' \in S_j \; s.t. \; a \equiv a'\}$
- $S_j^* = \{a' \in S_j : \nexists\, a \in S_i \; s.t. \; a \equiv a'\}$
- $S_{ij}^\cap = \{rep(a) : a \in S_i, \exists\, a' \in S_j \; s.t. \; a \equiv a'\}$

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

Example 7 With reference to Table 1, consider the merge between two product schemas, i.e., $S_1 \sqcup_{f_1} S_3$, and consider two records s and t, $s \in C_4$ with schema S_1, $t \in C_6$ with schema S_3. Let the values of ProductName be $s[a_6]$ = "Blueseventy Vision Goggles" and $t[a_8]$ = "B70 VG". The merge of s and t produces a record r where $r[a_6] = \mathbb{M}'(s[a_6], t[a_8])$ and $\mathbb{M}'$ is a conflict-resolution function that decides between "Blueseventy Vision Goggles" and "B70 VG".

Ultimately, the merge operator is applicable to two collections C_i and C_j represented by the same entity (i.e., $E = gran(C_i) = gran(C_j)$) and it is declared as $C_i \sqcup_{key(E)} C_j$. We further impose that the two collections share the same collection graph (i.e., $CG_{C_i} = CG_{C_j}$); this limitation is necessary (i) to enable the application of the merge operator to all entities in the two collections and (ii) to return a collection C_{ij} with the same representation of the two input collections. In particular, depending on the representation of the collection graphs, we distinguish the following three situations.

- If NoR, the result of the merge operation is trivial, as it simply produces a consistent recordset of E.
- If NeR (i.e., it is $E' \rightarrow E$, where E' is nested within E), the merge operation is recursively applied to the nested levels of the two collections. As per its definition, the merge on $key(E)$ produces a single attribute for the primitive attributes in common between the two collections, while the array attributes a_i and a_j respectively contain the records of E' in C_i and C_j are kept distinct. Thus, the merge operation is also applied between the recordsets within a_i and a_j to produce a consistent recordset of E' within a unique array a_{ij}. This is recursively applied if there is another entity $E'' \rightarrow E'$, where E'' is nested within E'.
- If FlR (i.e., it is $E \rightarrow E'$ and the records contain attributes of both E and E'), the merge cannot be applied, because each collection contains multiple records for the instances of E' (with potentially conflicting values). Indeed, this operation would need to be expanded to a complex set of operations, where the instances of E' are extrapolated from the respective collections, merged to resolve potential conflicts, and then joined back with the respective instances of E (see Forresi et al. 2021 for further details).[7]

[7] Ultimately, the same set of operations corresponds to converting the two collections into two separate NoR representations and merging

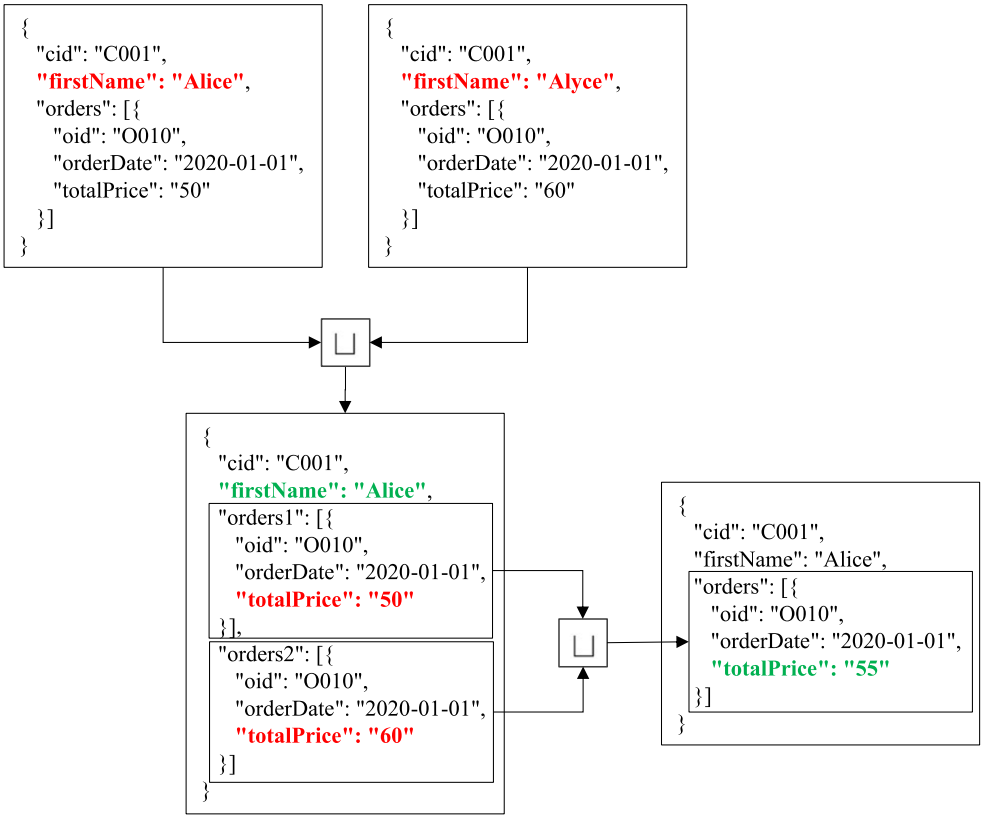

Fig. 8 Examples of merge operation on two records in NeR

Example 8 Figure 8 shows the effects of the merge operation on two records in NeR with $E_{or} \rightarrow E_{cu}$. First, the merge operation produces a consistent representation of E_{cu}; since two attributes with the same name cannot coexist, the orders arrays are renamed to orders1 and orders2. Then, the merge is applied on the two arrays to produce a consistent representation of E_{or} as well. Figure 9 shows an example of two records in FlR with $E_{or} \rightarrow E_{cu}$ that cannot be merged as resolving conflicts of E_{cu} instances in the same collection would require a complex set of operations.

5.2 Entity Views

To simplify the discussion on query plans, we introduce the notion of entity views. An entity view (EV) is a runtime-computed collection that provides a standard representation for the records modeling a set of entities. It is called a *view* because it is not persisted; differently from a typical view, however, it is not exposed to the user, but it is used internally to generate query plans.

them. Since the query planner in Section 6 explores all conversions to different representations, the restriction of the merge operator to collections in NeR and NoR does not limit the expressiveness of the approach.

Being a collection, an EV is associated with a collection graph as well, but only in one of the three special patterns (i.e., EVs can be only in either normal, nested or flat form).

Definition 11 (Entity view) An entity view is a collection χ whose records represent the features of a given set of entities in accordance to a schema representation. Its collection graph CG_χ is such that $rep(CG_\chi) \in \{NoR, NeR, FlR\}$.

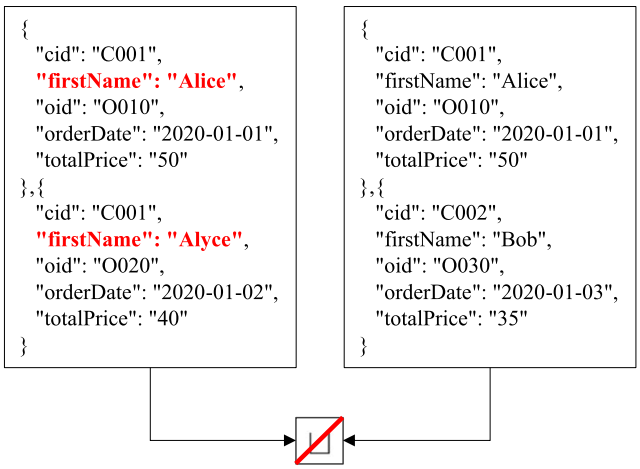

Fig. 9 Examples of two records in FlR that cannot be merged

An EV is either local or global. A *local* entity view (LEV) is obtained from collections belonging to the same database, thus it provides a partial representation of a set of entities. A *global* entity view (GEV) provides a complete and cleansed representation of a set of entities in the multistore; it can be obtained in two ways: either by merging LEVs represented by the same set of entities across *all* databases or through a join of GEVs.

Limiting EVs to the three schema representations poses certain limits; for instance, collection C_j in Fig. 7 mixes different representations, thus it cannot be considered an EV without applying first some schema transformations. However, enforcing these schema representations enables the definition of an algebra of operations on the EVs. In particular, operations on entity views are defined through *EV operators*, i.e., macro-NRA operators (distinguished from simple ones by the hat $\hat{}$ symbol) that embed a tree of NRA operations. Macro-NRA operators are high-level operators that aid the creation and discussion of execution plans.

5.2.1 LEV Creation

A LEV creation is defined as $\hat{\pi}(\mathcal{C}_\chi, CG_\chi, F_\chi, p_\chi)$. This operation creates a LEV χ defined by CG_χ from a set of collections $\mathcal{C}_\chi$ belonging to the same database. Notice that $\hat{\pi}$ also embeds the optional application of selection predicates p_χ; each $p \in p_\chi$ is in the form (f, ω, v), where f is a feature, $\omega \in \{=; >; <; \neq; \geq; \leq\}$ and v is a value. The underlying tree of NRA operations mainly depends on how the collection graphs of the collections in $\mathcal{C}_\chi$ differ from $rep(CG_\chi)$. The complexity of the creation operation is non-trivial and it is discussed in full detail in the Appendix. Figure 10 shows some examples.

- On the left the operations to obtain χ', i.e., a NeR view of E_{cu} and E_{or} from the relational collections C_1 and C_2. This requires to join the two collections (which produces a flat representation of E_{or} and E_{cu}) and then to change the collection's granularity to E_{cu} by nesting the order records within an array attribute (i.e., orders), created for each customer. Conversely, no operation would be necessary to obtain such a view from C_5, which is natively in NeR.
- On the right the operations to obtain χ'', i.e., a NoR view of E_{cu} from the document-based collection C_5. This requires to unnest the orders array attribute and to project only the features corresponding to E_{cu}. Conversely, no operation would be necessary to obtain such a view from C_2, which is natively in NoR.

5.2.2 GEV Creation

A GEV creation is defined as $\hat{\sqcup}(X, p_X)$, where X is a set of LEVs, $|X| \geq 2$, and p_X is an optional conjunction of selection predicates defined as in Section 5.2.1. This operation creates a GEV χ' by resolving conflicts between duplicated records from two or more LEVs sharing the *same* collection graph CG', i.e., $\forall \chi \in X$ it is $CG_\chi = CG'$. The details of the NRA plan produced by the GEV creation are discussed in the Appendix. Essentially, this macro-operator produces a left-deep tree of binary merge operations between LEVs. Once all LEVs have been merged, the optional selection predicates are applied.

5.2.3 Join of GEVs

A join of GEVs is defined as $\hat{\bowtie}(X)$, where X is a set of GEVs, $|X| \geq 2$. The obtained GEV χ' is the result of join operations between the GEVs in X, all of which represent connected but non-overlapping sets of entities: on the one hand, given any two entity views $\chi_i \in X$ and $\chi_j \in X$, it is $\mathcal{E}_{\chi_i} \cap \mathcal{E}_{\chi_i} = \varnothing$; on the other hand, the sets of entities must be connected, i.e., $\mathcal{E}_\chi = \bigcup_{\chi_i \in X} \mathcal{E}_{\chi_i}$ is a connected set in $\mathcal{D}_q$. The details of the NRA plan produced by the entity view join are discussed in the Appendix. Similar to the merge, this macro-operator produces a left-deep tree of binary join operations between two entity views. The result is a GEV that provides a cleansed representation of all the records in the multistore that are required to answer the query.

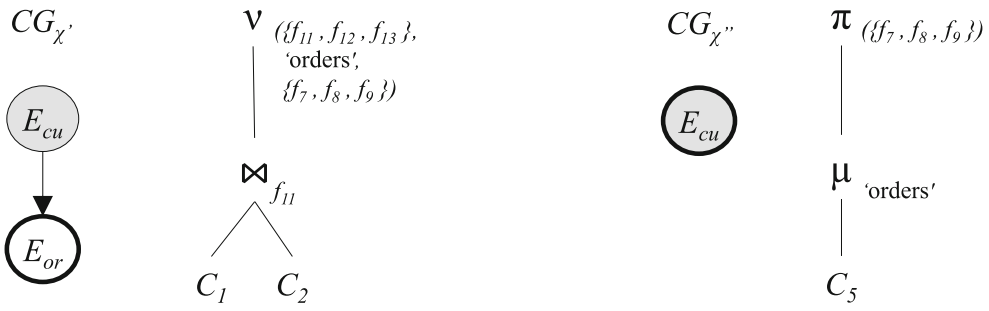

Fig. 10 Examples of NRA operations corresponding to LEV creations

5.2.4 Optimization of Entity View Operations

The three entity view operations defined above implement some logical rules to produce an optimized NRA tree.

- Despite the possible presence of multiple schemas in the same collection, the plans avoid accessing the same collection multiple times (unlike Forresi et al. 2021). In particular, mappings are exploited to produce predicates that take schema heterogeneity into consideration; for instance, a selection predicate on a feature f is translated to a disjunction of selection predicates on the attributes of $attr(f)$ that appear in the schemas of the collection.
- Predicate push-down is employed to apply selection predicates as close to the source as possible.
- Column pruning is employed to project from each collection the attributes corresponding to features that are relevant for the query.
- Whenever there is a need for joining (or merging) more than two collections (or entity views), a minimum selectivity heuristics (Steinbrunn et al., 1997) is adopted to determine the order of these join/merge operations. The basic idea is to start from the collection with the lowest cardinality and progressively merge it with collections with increasing cardinality. To estimate cardinalities we take into consideration the selection predicates and the statistics collected 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 value independence, and join containment.

6 Query Planning

The multistore supports the formulation of GPSJ queries, consisting of a group-by set (i.e., the features used to carry out an aggregation), one or more numerical features to be aggregated by some function (e.g., sum, average), and (possibly) some selection predicates.[8]

Definition 12 (Query) A GPSJ query q is a triple as $q = (q_\pi, q_\gamma, q_\sigma)$, where: q_π specifies the optional set of features to be projected; q_γ specifies optional aggregations as a set of pairs (f, op), where f is a numerical feature 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 is a feature, $\omega \in \{=, >, <, \neq, \geq, \leq\}$ and v is a value. At least one amongst q_π and q_γ must be defined.

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 9 Let q be the query that computes, for each customer, the average price of 2020 orders. The group-by set of q is $q_\pi = \{f_{11}\}$; the aggregation set is $q_\gamma = \{(f_9, avg)\}$ and the selection predicate set is $q_\sigma = \{(f_{10}, \geq,$ "2020/01/01")$\}$.

Building the 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, computing the average price for each customer requires to access not only E_{cu} and E_{ol} but also E_{or}, even if no feature belonging to E_{or} is mentioned in the query. Thus, we define the *query graph* as a conceptual view of the query on the dataspace.

Definition 13 (Query graph) The query graph QG_q is an acyclic subgraph of $\mathcal{D}$ (i.e., $QG_q = (\mathcal{E}_q \subseteq \mathcal{E}, L_q \subseteq L)$) such that:

(i) QG_q is minimally connected;
(ii) $\mathcal{E}_q \supseteq attr(q)$;
(iii) $\exists\, E^* = fact(QG_q) : feat(E^*) \supseteq feat(q_\gamma)$.

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., there is a single fact E^* and the features in q_γ belong to $feat(E^*)$). Many subgraphs could exist for a given query since more *(many/one)-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 user interaction to identify the adequate subgraph. If no subgraph exists, the query is not answerable.

6.1 Enumeration of Query Plans

QG_q is the starting point to define the query plan, which can be defined in terms of entity view operations.

[8] Joins are implicitly defined, depending on the features selected in the group-by set, aggregation, and selection. Though this limits joins to the relationships discovered between entities in the dataspace, we remark that the pay-as-you-go approach enables user to refine the dataspace at any time.

Definition 14 (Query plan) A query plan P is a rooted tree of entity view operations, where (i) the root is a GEV join operation ($\hat{\bowtie}$), (ii) the root is preceded by one or more GEV creation operations ($\hat{\sqcup}$), and (iii) each of the latter is preceded by one or more (parallel) LEV creation operations ($\hat{\pi}$). The root is possibly extended with an NRA aggregation operation (γ).

Example 10 Figure 11 shows a sample plan for a query that computes, for each gender, the average quantities bought for products of brand "BrandABC"; operators' predicates are omitted for space reasons. In the upper part, two LEV creation operations compute an entity view in NeR with customers, orders, and order lines records from the collection in the document-based database (i.e., C_5) and the tables in the relational one (i.e., C_1 to C_3), respectively; in particular, the latter is the one hiding the most complexity, as multiple join and nest operations are required to compute the NeR representation. The two LEVs are then merged in a GEV creation operation, that returns a cleansed FlR representation of the same data and projects the only features required by subsequent operations. Similarly in the lower part, two other LEV creation operations compute an entity view in NoR with product records from C_6 and C_4, respectively. The subsequent GEV creation operation merges the products and applies the filter on the reconciled records. Ultimately, the GEV join operation combines the produced GEVs, while the aggregation operation computes the final result.

Several query plans can be devised for the same query. For readability purposes, the full enumeration process is detailed in the Appendix. The factors that determine the number of alternative query plans and that drive the enumeration process are summarized below.

#1 *LEV creation*. As discussed in Section 5.2.2, a GEV creation consists in left-deep trees of merge operations involving all LEVs with the same collection graph.

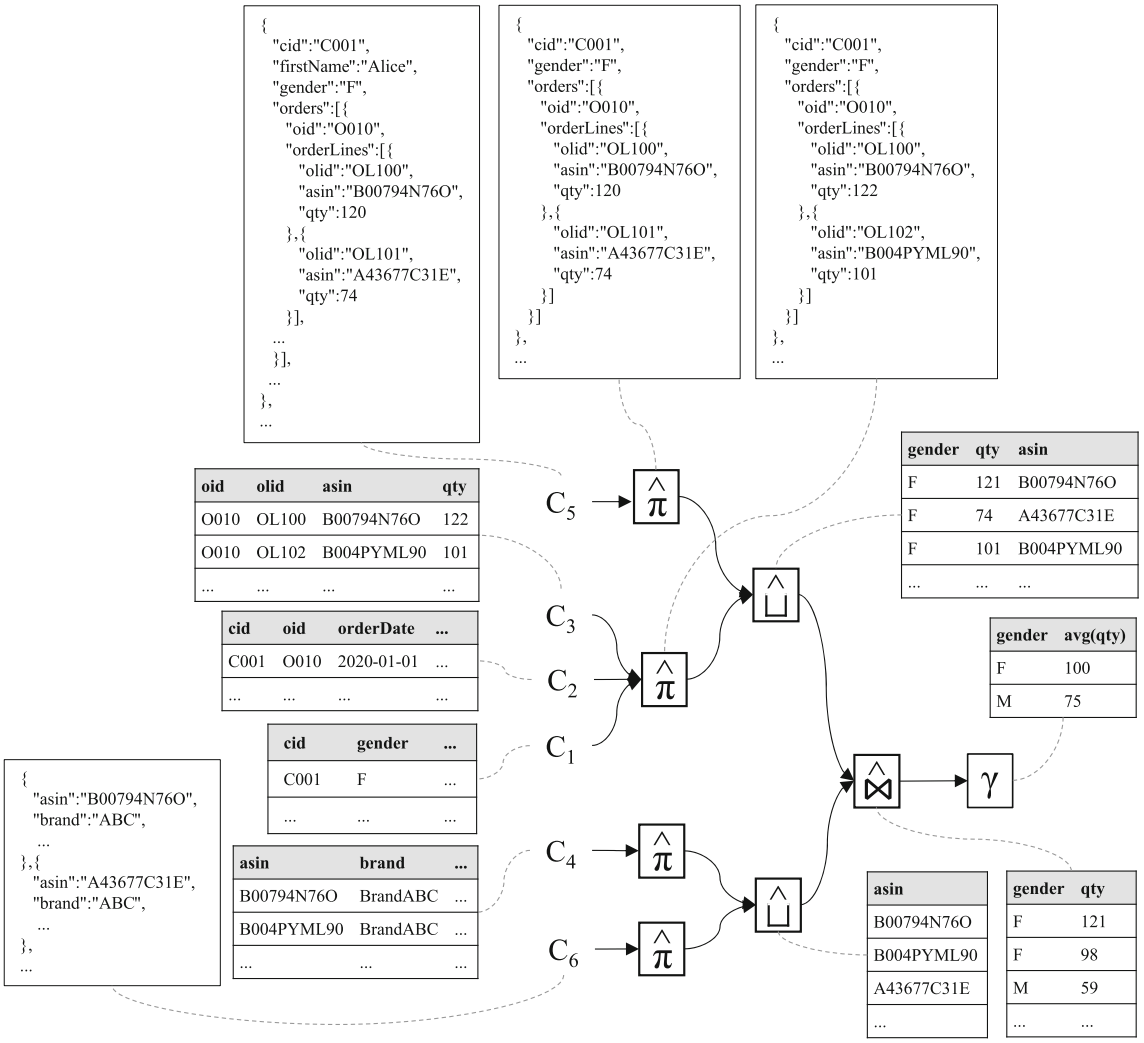

Fig. 11 Example of a full query plan. Data flows are represented by full arcs while dotted lines link collections and EV to sample data

Thus, different plans can be defined by choosing different schema representations to create the LEVs of a given GEV. Let χ a GEV with $CG_\chi = (\mathcal{E}_\chi, L_\chi)$.

- If $|\mathcal{E}_\chi| = 1$, then LEVs are created in NoR (by definition, only NoR supports a single entity in $\mathcal{E}_\chi$).
- If $|\mathcal{E}_\chi| > 1$, then LEVs can be created either in FlR or NeR, depending on the assumptions from Section 5.2: FlR is possible only if $\nexists E \in \mathcal{E}_\chi : \phi(E) = true$ (because the merge operation requires EVs in NoR or NeR); NeR is possible only if there is a single path of -to-one relationships in u (which is a structural requirement for the NeR pattern).

#2 *GEV creation*. A query plan may include several alternative combinations of GEVs. These combinations can be found by partitioning $QG_q = (\mathcal{E}_q, L_q)$ into one or more subgraphs and creating a GEV for each subgraph. The number of possible combinations is $2^{|L_q|}$. Fortunately, not all combinations are feasible: by definition, a GEV creation requires all corresponding LEVs to share the same collection graph, which is not always possible (see Example 11).

#3 *LEV allocation*. Each LEV creation operation can be executed either directly by the middleware or pushed down to the database storing the respective data; Section 7 will show that both options can be efficient, depending on several factors. As the computation of each LEV is independent of the others, the number of possibilities is 2^y, where y is the number of LEVs to be created.

Example 11 The boxes in Fig. 12 correspond to all possible query plans for the query graph on the left. For a given box, each subgraph corresponds to a GEV, where the coloring scheme defines its collection graph. Horizontally, the plans differ on the way the query graph is partitioned into GEVs (#2); vertically, the plans differ on the schema representation for the given GEVs (#1); factor (#3) is not shown in the figure. Plans with an orange background are unfeasibile: as shown in Fig. 3, there is no collection in the document-based database described by E_{pr} and there is no collection in the wide-column database described by E_{ol}. This makes it impossible to create a LEV χ such that $E_\chi \subseteq \{E_{pr}, E_{ol}\}$ on either database — which is a requirement to create the corresponding GEV.

7 Cost Model

Enumerating all the feasible query plans is useless if we are not able to identify the one determining the lowest execution time. Identifying the most efficient among the high number of plans obtainable for a single query q is particularly challenging in a multistore due to:

- *DBMSes heterogeneity*. In a multistore, several DBMSes coexist each with its own data model and capabilities. A multistore cost model must cover all the different DBMSes.
- *DBMS variability*. In a big data and pay-as-you-go context, variability is the norm. New databases may be added and DBMS resources (e.g., number of CPU cores, amount of RAM) could change along time.

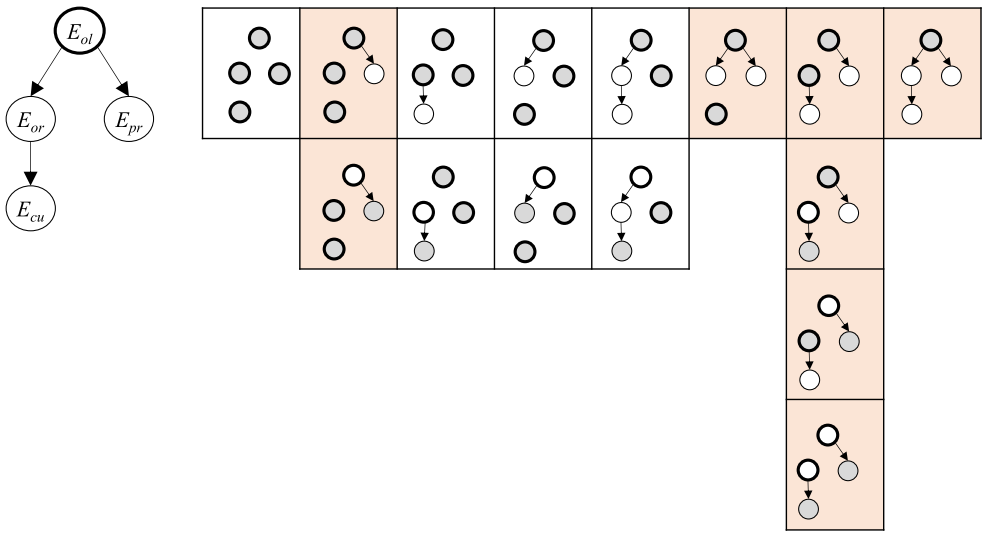

Fig. 12 A query graph (left) and all the possible plans, shown with the corresponding combinations of GEVs (right). Unfeasible plans are in orange; recall that bold circles correspond to fact entities

In Forresi et al. (2021), we relied on existing literature to model the cost of each NRA operation on each engine in terms of read and written disk pages. While this worked well on the simple example considered by Forresi et al. (2021), (i) it did not consider the amount of resources assigned to each engine, (ii) it made simplistic assumptions about the parallelization of the computation in a distributed engine, (iii) it considered execution costs related to disk I/O only, (iv) it required an advanced knowledge about the internal details of each engine and related algorithms that reduces its extensibility.

In this paper, we overcome these limitations by proposing a self-learning cost model, which implicitly captures the aforementioned aspects without requiring explicit and complex modeling of execution costs. Inspired by Golfarelli et al. (2019), the cost model is composed by a set of multi-regression models $H = \{h_0(), ..., h_n()\}$, one for each of the n execution engines composing the multistore including the middleware denoted by $h_0()$. The query plan P is partitioned in a set subplans SP, each corresponding to the execution of an entity view operation on an engine (including the middleware). A multi-regression model $h_{eng(P')}(P')$ estimates the execution time for the subplan P' on the corresponding engine $eng(P')$ based on a plan profile. The list of the features composing the profiles is reported in Table 3; for each feature, the domain and the supported engines are indicated (e.g., the number of join operations is not captured on wide-column engines since they do not support this operation). Some of these features are directly obtained from the plan (e.g., number of unnest operations embedded in a $\hat{\pi}$ or $\hat{\cup}$ operation), while others also require basic statistics on the local databases (e.g., the collections' cardinalities and attributes' histograms to compute selectivity and aggregation rate, the presence of indexes to infer whether the engine will be able to exploit them). The execution time for P is estimated by composing the execution time of its subplans SP as follows:

$$Time(P) = \sum_{P' \in SP | eng(P')=0} h_0(P') + \max_{i \in [1,n]} \sum_{P' \in SP | eng(P')=i} h_i(P')$$

The assumption in taking the maximum execution time of a single engine is that computations on the local databases are carried out concurrently (indeed, all subplans are launched at the same time from the middleware and are assumed to be executed sequentially within each engine). Thus, the cost model explicitly codes *inter-engine* parallelism by considering only the slowest engine as the one contributing to the plan duration. Conversely, we assume that the computation on the middleware requires all local computations to be completed first; this is a worst-case simplification, as – depending on the actual plan – the global computation could be started to process the results of the first local computation. Noticeably, *intra-engine* parallelism generated by multi-core execution is transparently captured by the regression models.

Figure 13 reports, for the multistore used in our tests, an excerpt of regression tree for the middleware (i.e. Spark) showing the most important features determining the execution time. The full regression tree includes 98 leaves arranged in 5 levels. Each leaf estimates the execution time limitedly to the portion of the feature space defined by the split nodes. We remind that the advantage of local regression

Table 3 The features of the regression models; the engine's legend is R=Relational, D=Document-based, W=Wide-column, K=Key-value, M=Middleware

Profile Feature	Domain	Engine supp.
Entity view operation	$\{\hat{\pi}, \hat{\cup}, \hat{\bowtie}\}$	R D W K M
Source schema representation	{NeR, NoR, FlR}	R D W K M
Target schema representation	{NeR, NoR, FlR}	R D W K M
Number of records	$\mathbb{N}$	R D W K M
Selectivity	[0, 1]	R D W K M
Aggregation rate	[0, 1]	R D - - M
Number of unnest operations	$\mathbb{N}$	R D - - M
Number of join operations	$\mathbb{N}$	R D - - M
Number of union operations	$\mathbb{N}$	- - - - M
Number of merge operations	$\mathbb{N}$	- - - - M
Number of selections exploiting indexes or source partitioning	$\mathbb{N}$	R D W K M
Number of selections not exploiting indexes or source partitioning	$\mathbb{N}$	R D W K M
Number of nested selections exploiting indexes	$\mathbb{N}$	R D - - M
Number of nested selections not exploiting indexes	$\mathbb{N}$	R D - - M
Number of aggregations (nest, group by)	$\mathbb{N}$	R D - - M

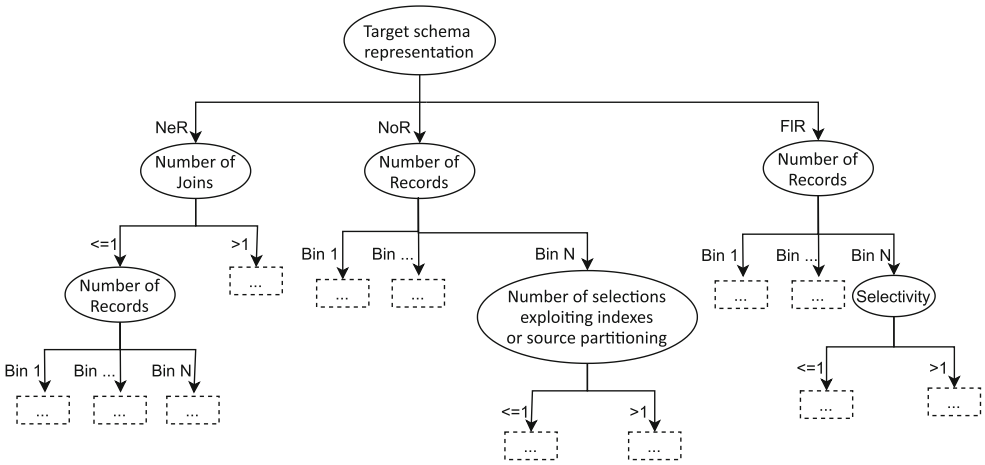

Fig. 13 An excerpt of the regression tree defining the cost model for the middleware engine

approaches (Baldacci et al. 2016; Loader, 2006) is that they enable the fitting of complex feature spaces without using global regression functions that are implicitly difficult to tune. To reach this goal, several simple regression functions, each modeling a small part of the data space, are combined to create a complex model.

Models' training consists in capturing, for each execution engine, the execution times of each entity view operation from a wide set of queries, manually defined by varying (i) the number of involved entities, (ii) the selectivity of the selection predicates, and (iii) the strength of the final aggregation (further details in Section 8.3). Entity view operations are executed in random order and repeated 5 times to smooth occasional outliers. Query profiles are then created and eventually pre-processed: binning is applied to features with a wide range of values (i.e., "Selectivity", "Aggregation rate", and "Number of records"); one-hot encoding is used to convert categorical features to numbers; features with a low range of values are not modified (i.e., all "Number of *operation*" features). Finally, the regression induction algorithm proposed is fed with the training sets to create the regression models.

Models' drift (typically due to the variation in the amount of engine resources) is detected when the absolute relative error on the estimate becomes higher than a given threshold. The training queries for the drifted engine are re-executed and the corresponding regression tree is rebuilt. Similarly, if a new database is added to the multistore, a new set of queries must be devised and an additional regression tree must be trained.

8 Experiments

8.1 Prototype

The 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 databases, respectively. PostgreSQL is installed on a single machine, while NoSQL stores are distributed across 5 machines. The middleware (including the query planner, the cost model, and the execution framework) is implemented as an Apache Spark application; Spark is one of the most used open-source execution frameworks for Apache Hadoop clusters, providing connectors to most databases (including those in our multistore). The Spark application runs with 4 executors, each with 4 CPU cores and 8GB RAM.

The user interacts with the multistore through a set of REST APIs to visualize and maintain the dataspace and to submit GPSJ queries. Plain Scala is used to generate the regression models for each execution engine (by relying on Weka libraries), as well as to build the query graph (by relying on the Graph for Scala library) and to generate the execution plans in sub-second time (Forresi et al., 2021). Queries are formulated by relying on the SQL APIs exposed by Spark's DataFrame abstraction in two steps: (i) a DataFrame is initialized for each subplan assigned to a local database by making a custom call to the database systems' APIs, thus overcoming the current limitations of Spark that prevent the push-down of a custom set of operations (Delaney & Li, 2017); (ii) the subplans assigned to the middleware are formulated through the SQL APIs, where User Defined Functions (UDFs) have been defined to implement the NRA extensions.

8.2 Multistore Benchmarks

The multistore in Fig. 3 is a variation of the multi-model benchmark Unibench (Zhang et al., 2018), which has been extended to inject schema heterogeneity and introduce

overlapping records in different databases. Since the choice of the best query plan is determined by several factors (see Section 7), different benchmarks of the same multistore have been generated. In particular, the benchmarks vary on the following parameters.

- Overlap: the presence/absence of overlap for different entities determines different workloads with different execution plans; as the preliminary results by Forresi et al. (2021) had shown, FlR is favored in absence of overlap, while NeR is favored in its presence. The overlap parameter defines, for each benchmark, the set of entities $\mathcal{E}_\phi$ for which data fusion activities must be carried out (i.e., $\forall\ E \in \mathcal{E}_\phi$ it is $\phi(E) = true$). The overlap rate is set to 60% for E_{pr}, and to 20% for both E_{or} and E_{cu}.
- Data skewness: the convenience of pushing-down computations to local databases is partly determined by the amount of the data that must be processed by each database. Thus, different benchmarks are created with an unbalanced distribution of records, so as to put more pressure (i.e., more data) on different databases. Given the query expressiveness limitations of Cassandra and Redis (where no significant operations can be pushed-down), this variation is applied on MongoDB and PostgreSQL. Data skewness is determined by features f_{10} and f_{14}, so as to enable the formulation of query predicates that select different amounts of data from either database.

The summary of the generated versions is shown in Table 4. Remarkably, the main goal of the paper is not to prove the scalability of the multistore (which is already addressed by Forresi et al. (2021)) but rather to evaluate the performance of the different execution plans in reasonable times. Nonetheless, all multistores are deployed with a scale factors 1, 10, and 100 to consider scalability as well. Scale factor 1 determines a total of 300K order lines relative to 10K customers and 10K products, which offers a good trade-off between a sufficient cardinality and non-excessive execution times for the experimental evaluation. The data used to populate the collections in the different databases is available at https://big.csr.unibo.it/multistore.

Table 4 Generated multistore benchmarks with different entity overlapping settings and different data skewness

Multistore benchmark	Overlap	Data skewness
MS_1	$\mathcal{E}_\phi = \{E_{pr}, E_{cu}\}$	R
MS_2	$\mathcal{E}_\phi = \{E_{pr}, E_{ol}, E_{or}, E_{cu}\}$	R
MS_3	$\mathcal{E}_\phi = \{E_{pr}\}$	R
MS_4	$\mathcal{E}_\phi = \{E_{pr}, E_{cu}\}$	D
MS_5	$\mathcal{E}_\phi = \{E_{pr}, E_{ol}, E_{or}, E_{cu}\}$	D
MS_6	$\mathcal{E}_\phi = \{E_{pr}\}$	D

If data skewness is R, 90% of the customer, order, and orderline records are stored on PostgreSQL, and the remainder on MongoDB; it is the opposite when data skewness is D

8.3 Cost Model Evaluation

The tests in this section are aimed at assessing the quality of the cost model in choosing the query plan for a given query. Given the expressiveness of GPSJ queries (Golfarelli et al., 1998), the workload has been devised by varying the queries in terms of selectivity, group-by strengths, and number of entities involved in the query.[9]

- 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).
- The selection predicate is either absent, weak (i.e., its selectivity is low; for instance, a filter on a year of data), or strong (i.e., its selectivity is high; for instance, a filter on a given date).
- The number of entities involved in the query varies from 1 to all 5 of them.

By varying these components, we obtain 54 queries, 48 of which are used to train the cost model and the remaining ones for testing. Details about all queries are available at available at https://big.csr.unibo.it/multistore. Overall, the estimated execution times differ from the actual ones for 9.4% on average; this is sufficient to correctly estimate the best execution plan for a query.

Figure 14 shows the average number of plans generated by the cost model with increasing number of entities involved in the query. As discussed in Section 6.1, the enumeration of query plans is subject to exponential factors that translate into an increasing (but non-exponential) numbers of plans generated as more entities are queried. Nonetheless, their enumeration does not have a great impact on performance: besides the time required for the query graph generation and partitioning (75ms), the generation and cost estimation of a single plan accounts for $2\tilde{m}s$. Overall, the cost model always runs in sub-second times, with an average of 210ms.

The effectiveness of the cost model is measured against 5 selected baseline strategies:

[9] Producing representative analytical workloads is still a research challenge to this day (Lu & Holubová, 2019). Nonetheless, several proposals in the related literature adopt criteria similar to ours in the definition of representative workloads (Golfarelli & Saltarelli, 2003; Darmont et al., 2005; O'Neil et al., 2009; Bimonte et al., 2021).

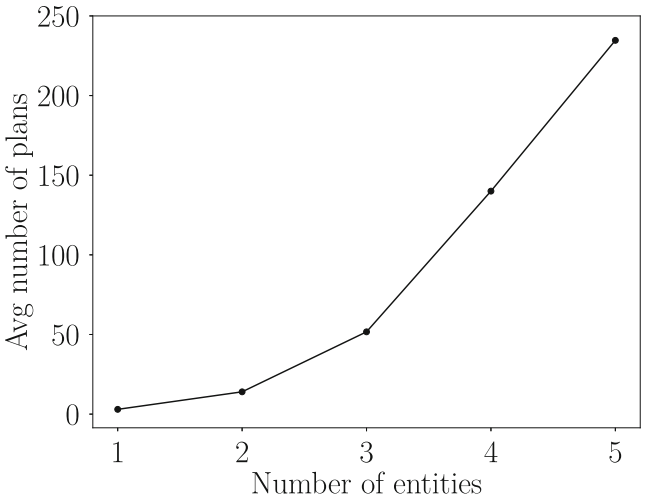

Fig. 14 Average number of enumerated plans with increasing number of queried entities

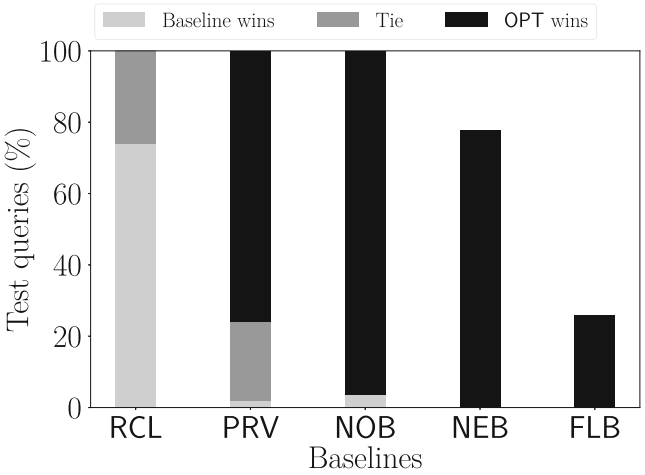

Fig. 15 Pair-wise comparison between plan selection strategies

- RCL corresponds to the hypothetical oracle that always chooses the optimal plan;
- PRV corresponds to our previous multistore implementation (Forresi et al., 2021), where LEVs were always created in NoR and the computation was not pushed down to the databases;
- NOB, NEB, and FLB adopt a simple strategy to choose the plan that maximizes both computation push-down and the creation of LEVs in a given schema representation (respectively NoR, NeR, and FlR).

OPT refers to the optimized multistore discussed in this paper. Table 5 indicates the average time increase between the execution time of the plan chosen by each strategy and the one of the optimal plan (RCL is not included here, as the time increase would obviously be 0). The data shows that OPT outperforms every other baseline, consistently throughout all benchmarks. More in detail, Fig. 15 separately compares OPT with every baseline to determine how many times OPT chooses a plan that is better than the one chosen by the compared baseline. Again, OPT emerges as a clear winner against all baselines, especially proving a big step forward with respect to Forresi et al. (2021); the bars of FLB and NEB do not reach 100% due to the impossibility to create LEVs in FlR or NeR in some queries (e.g., those involving a single entity) or benchmarks (as FlR is incompatible with the merge operation).

The percentage of ties between OPT and RCL (i.e., 25.9%) indicates there is room for improvement. However, further investigation reveals that OPT mostly returns sub-optimal plans when the optimal one is missed. This is confirmed in Fig. 16, which shows, for each strategy, the percentage of times that the chosen plan is among the top-K most efficient ones; it is observable here how OPT reaches a high accuracy more quickly than other baselines.

Figure 17 shows how the overhead of OPT and the baselines increases with the increase in scale factor. Noticeably, overheads tend to be higher at higher scale factors; this is mainly due to a performance degradation on the RDBMS. Remarkably, though OPT is affected by such increase, it improves its advantage over all baselines.

Finally, Fig. 18 reports the average overhead of OPT with respect to different levels of training completeness, i.e., by progressively depriving the cost model of the collected training data. Essentially, this corresponds to testing the accuracy of the cost model with queries that are increasingly more distant from the training queries. The reported values are averaged from 5 experiments where different random samples of data are removed. The experiment shows that

Table 5 Average overhead of the chosen plan with respect to the optimal one

	OPT	PRV	NOB	NEB	FLB
MS_1	**21.4%**	48.1%	143.5%	84.2%	–
MS_2	**19.6%**	39.4%	131.9%	85.5%	–
MS_3	**7.7%**	59.1%	192.0%	120.1%	113.6%
MS_4	**49.0%**	122.1%	202.6%	114.3%	–
MS_5	**20.7%**	27.7%	176.6%	141.7%	–
MS_6	**6.4%**	53.1%	365.2%	171.2%	176.5%
AVG	**20.8%**	58.3%	202.0%	119.5%	145.1%

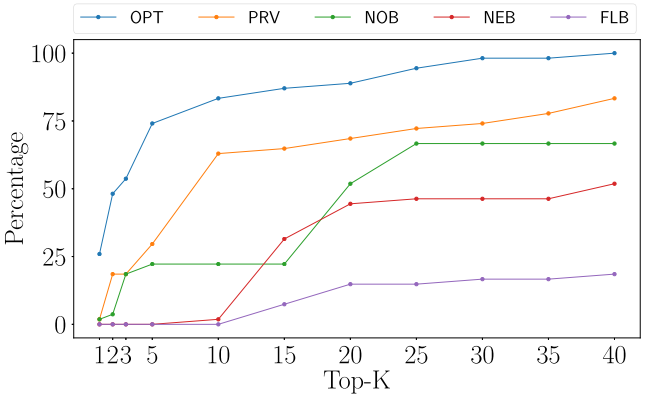

Fig. 16 Percentage of plan chosen among the top-K for every plan selection strategy

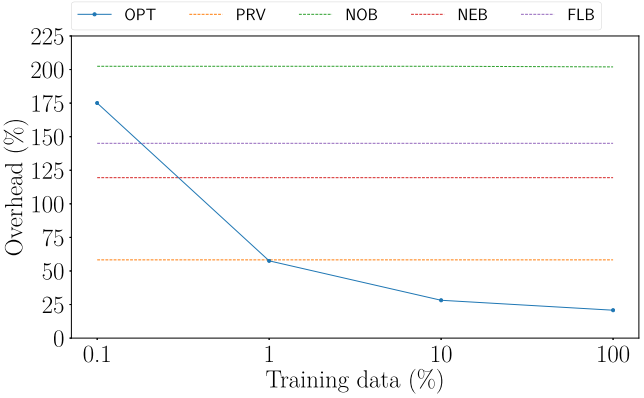

Fig. 18 Average overhead with different levels of training completeness

OPT is able to rapidly converge to a low overhead and that it is able to provide good effectiveness even when the training data is very limited.

8.4 Query Plans Evaluation

The tests in this section analyze the factors that impact on query performance. For this purpose we collected the execution times of every possible execution plan generated by the query planner in the same set of queries from Section 8.3.

Figure 19a shows the average execution time decrease (in %) by varying the percentage of operations pushed down to MongoDB (D) and PostgreSQL (R). The decrease is measured against the execution time in absence of push-down. Given a plan P, let SP_{D} and SP_{R} be the set of subplans corresponding to LEV creation operations where the data is taken from the document-based and the relational database, respectively; also, let $nra()$ be a function that returns the number of NRA operations in a given subplan. Then, the x and y coordinates of P in Fig. 19a respectively correspond to $\frac{nra(\{P' \in SP_{\mathrm{R}} : eng(P')=\mathrm{R}\})}{nra(SP_{\mathrm{R}})}$ and $\frac{nra(\{P' \in SP_{\mathrm{D}} : eng(P')=\mathrm{D}\})}{nra(SP_{\mathrm{D}})}$. Figure 19b shows the average execution time (in seconds) at different amounts of data (in MB) transferred from the local database (MongoDB above, PostgreSQL below) to Spark. A thorough evaluation of the single execution plans allowed us to obtain the following findings.

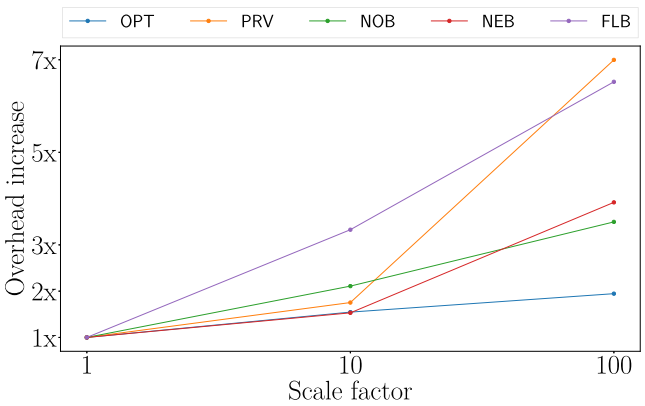

Fig. 17 Overhead increase with increasing scale factors

- Push-down on MongoDB emerges as a generally convenient solution. The main reason is found in (i) reducing the amount of data moved from MongoDB to Spark, and (ii) Spark operating less efficiently on nested data. First, applying filters at the middleware level implies transferring a (potentially) large amount of data that will be discarded later; this is particularly true in the application of filters to nested attributes, as pushing down this operation accounts for an average time decrease of 33%. Figure 19b shows that there is a correlation between the amount of moved data and the execution time. Additionally, we observe that MongoDB is much faster than Spark in handling unnest operations (required to compute LEVs in NoR or FlR from the nested collection C_5) and in projecting features out of nested attributes. The latter is especially observable in plans where the computation pushed down to MongoDB is minimum (because computing the LEV in NeR from C_5 for E_{cu}, E_{or}, and E_{ol} requires no extra computation besides the projection of features from attributes), yet there is an average 24% decrease with respect to carrying out the same operations on Spark.
- No clear trend emerges on PostgreSQL. On the one hand, the push-down of computation implies less data movement and a better exploitation of indexes and statistics; on the other, Spark is a more powerful engine and is able to exploit operation pipelining to minimize the amount of data shuffling. Differently from MongoDB, Fig. 19b shows that execution times are not directly correlated to the amount of moved data.

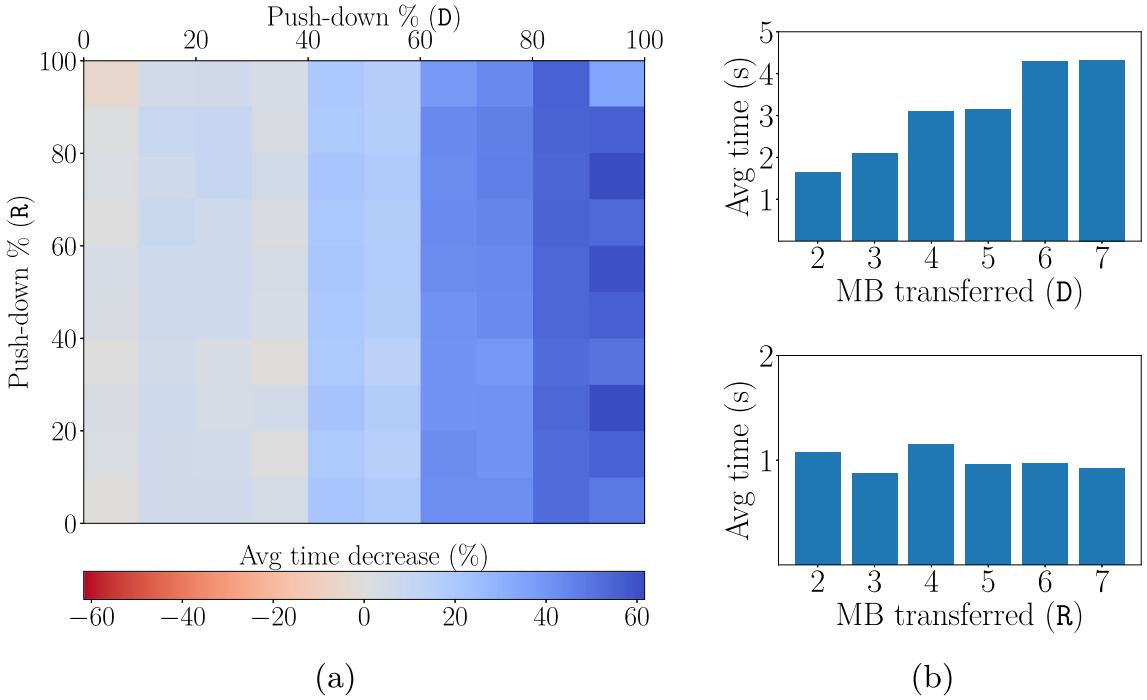

Fig. 19 On the left, the average execution time decrease (in %) at different percentages of operations pushed down to PostgreSQL (R) and MongoDB (D); the decrease is measured against the execution time in absence of push-down (i.e., the bottom-left corner, measuring 0%). On the right, the average execution time (in seconds) at different amounts of data (in MB) transferred from the local database (MongoDB above, PostgreSQL below) to Spark

Interestingly, we have observed a performance decrease in query plans with LEVs on relational data where a large amount of operations is split between the database and the middleware (however, this is not reflected in the figure due to the averaging with other query plans).

The results in Fig. 19a depend on the resources available to each execution engine. In Fig. 20 we measure this dependency by reducing the amount of resources to the local databases: in particular, we move PostgreSQL to a single-core (versus the original 8-core) machine and we consider a non-sharded implementation of MongoDB's collections (versus the sharding on 5 machines). The results show, for each database, the average time increase (in %) when running the same sets of plans on the less-powerful version of such database. As expected, pushing computation down becomes a less-favorable choice. Notice that execution times increase even at 0% of computation push-down; that is because data still needs to be read from the database (which is obviously slower in the less-powerful version).

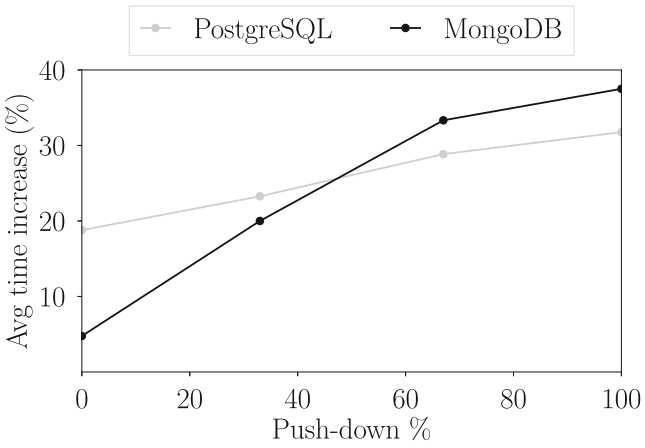

Fig. 20 Average execution time increase with fewer resources assigned to PostgreSQL or MongoDB

A final consideration concerns the usage of different schema representations in LEV creation operations. Interestingly, in 86% of the queries executed on benchmarks with most data on the document-based database (i.e., MS_4 to MS_6), the best query plan mostly relies on NeR to create LEVs. In the other benchmarks, the schema representation most adopted by the best query plan also depends on the level of overlap: in case of medium/high overlap (i.e., MS_2 and MS_3), NeR is chosen for 87% of the queries; in case of low overlap (i.e., MS_1), NoR and FlR are chosen for 75% of the queries. These results are consistent with previous findings (Forresi et al., 2021) stating that (i) the need to solve record overlapping influences the choice of the schema

representation, and (ii) preserving the original modeling of data usually translates to faster executions. Ultimately, the absence of a clear winner is a further validation of the need of a cost model to choose an efficient execution plan.

9 Conclusions

In this work, we have proposed a cost-based optimization of execution plans in a multistore by devising and evaluating different strategies to carry out joins and data fusion in presence of data replication. The execution plans are generated in terms of a multistore algebra extended from NRA and are based on different schema representations, so as to possibly take advantage of the original modeling of the data in the local databases. Experiments have been carried out on different multistore benchmarks to investigate the factors that drive the performance of different execution plans, demonstrating that the variety of these factors motivates the evaluation of alternative plans. The plan to execute for a given query is chosen by a black-box self-learning cost model; the effectiveness of the latter is shown by the experiments, which demonstrate the success in optimizing query executions with respect to Forresi et al. (2021).

Future work is planned to take several paths. First, we plan to evolve the multistore into a more complex data platform extending functionalities from simple querying to data search and profiling, provenance investigation, and orchestration of application pipelines (Francia et al., 2021). On the other hand, the efficiency of our multistore system can be further improved by considering the addition of data aggregation push-down to the local databases. This translates to the creation of new execution plans that need to be evaluated by the cost model as well. In this direction, the findings of this paper concerning the factors that drive execution plans' performance can serve as a basis to define rules or heuristics that reduce the number of generated execution plans. Finally, we plan to complete the multistore's functionalities by adding support to the graph data model.

Appendix: Query Plan Algorithms

This section details the algorithms that are used to create query plans. In these algorithms, functions $addEvOp$, $addUnOp$, and $addBinOp$ return a plan that extends the one given in input with the EV (or NRA) operation specified in the same input. Also, function $graph(P)$ returns the collection graph of the collection returned by P.

A.1 Query Plan Enumeration

Algorithm 1 details the procedure to define query plans. A query plan is defined for every feasible partitioning U of the query graph QG_q (Lines 3 to 30); a GEV is defined for every partition u (Lines 5 to 23); a LEV is defined for every database containing records of the entities in $\mathcal{E}_u$ (Lines 7 to 18). In Lines 9 to 14 the parameter for the LEV creation operation are retrieved and used in Line 15 to define such operation; in particular, Line 13 selects the entities of the LEV for which there is no record overlap (i.e., $\phi(E) = false$), so as to push-down to the $\hat{\pi}$ operation the selection predicates on features of such entities (Line 14). Once $\mathcal{P}_{LEVs}$ is created as the set of all plans to produce the LEVs of a given GEV, the GEV creation operation is defined (Lines 9 to 14); symmetrically, Line19 selects the entities of the GEV for which there is record overlap (i.e., $\phi(E) = true$), so as to apply the selection predicates on features of such entities (Line 20) to the $\hat{\sqcup}$ operation. Finally,

Require: $q = (q_\pi, q_\gamma, q_\sigma)$: the query; $QG_q = (\mathcal{E}_q, L_q)$: the query graph.
Ensure: $\mathcal{P}$: the set of query plans.
1: $\mathcal{P} \leftarrow \emptyset$
2: $\mathcal{U} \leftarrow getFeasiblePartitionings(QG_q)$
3: **for all** $U \in \mathcal{U}$ **do** ▷ A logical plan for each partitioning
4: $\quad\mathcal{P}_{GEVs} \leftarrow \emptyset$
5: $\quad$**for all** $u \in U, u = (\mathcal{E}_u, L_u)$ **do** ▷ A GEV for each partition
6: $\quad\quad\mathcal{P}_{LEVs} \leftarrow \emptyset$
7: $\quad\quad$**for all** $db \in DB$ **do** ▷ A LEV for each database
8: $\quad\quad\quad P \leftarrow$ **new** $PlanNode()$
9: $\quad\quad\quad\mathcal{C}_\chi \leftarrow \{C \in collections(db)\ s.t.\ \mathcal{E}_C \cap \mathcal{E}_u \neq \emptyset\}$
10: $\quad\quad\quad$**if** $\mathcal{C}_\chi \neq \emptyset$ **then**
11: $\quad\quad\quad\quad\mathcal{E}_\chi \leftarrow \bigcup_{C \in \mathcal{C}_\chi} \mathcal{E}_C$
12: $\quad\quad\quad\quad F_\chi \leftarrow \{feat(q_\pi) \cup feat(q_\gamma) \cup F_\sqcup\} \cap feat(\mathcal{E}_\chi)$
13: $\quad\quad\quad\quad\mathcal{E}'_\chi \leftarrow \{E \in \mathcal{E}_\chi\ s.t.\ \phi(E) = false\}$
14: $\quad\quad\quad\quad p_\chi \leftarrow \{p \in q_\sigma, p = (f, \omega, v)\ s.t.\ f \in feat(\mathcal{E}'_\chi)\}$
15: $\quad\quad\quad\quad P \leftarrow addEvOp(P, \hat{\pi}, (\mathcal{C}_\chi, CG_u, F_\chi, p_\chi))$ ▷ LEV creation
16: $\quad\quad\quad\quad\mathcal{P}_{LEVs} \leftarrow \mathcal{P}_{LEVs} \cup P$
17: $\quad\quad\quad$**end if**
18: $\quad\quad$**end for**
19: $\quad\quad\mathcal{E}'_\chi \leftarrow \{E \in \mathcal{E}'_\chi\ s.t.\ \phi(E) = true\}$
20: $\quad\quad p_\chi \leftarrow \{p \in q_\sigma, p = (f, \omega, v)\ s.t.\ f \in feat(\mathcal{E}'_\chi)\}$
21: $\quad\quad P \leftarrow addEvOp(\mathcal{P}_{LEVs}, \hat{\sqcup}, (p_\chi))$ ▷ GEV creation
22: $\quad\quad\mathcal{P}_{GEVs} \leftarrow \mathcal{P}_{GEVs} \cup P$
23: $\quad$**end for**
24: $\quad P \leftarrow addEvOp(\mathcal{P}_{GEVs}, \hat{\bowtie})$ ▷ GEV join
25: $\quad$**if** $q_\gamma \neq \emptyset$ **then**
26: $\quad\quad predicate \leftarrow (q_\pi, q_\gamma)$
27: $\quad\quad P_q \leftarrow addUnOp(P_q, \gamma, predicate)$ ▷ Final group by
28: $\quad$**end if**
29: $\quad\mathcal{P} \leftarrow \mathcal{P} \cup P$
30: **end for**
31: **return** $\mathcal{P}$

Algorithm 1 planGeneration(q, QG_q)

the GEV join is defined (Line 24), possibly followed by the final aggregation operation, if specified in the query (Lines 25 to 27).

The algorithm omits the choice of the execution engine for each LEV: given a logical plan P with n LEVs, 2^n physical plans are generated with all combinations of assigning, to each $\hat{\pi}$ operation, either the middleware or the local database.

A.2 From EV Operations to NRA Operations

Algorithms 2 to 5 detail how each EV operator defined in Algorithm 1 is translated into a plan of NRA operations.

Require: $\mathcal{P}_X$: a set of NRA plans to compute the GEVs X; QG_q: the query graph.
Ensure: P: the NRA plan that joins the GEVs together.
1: $\mathcal{P}_X = sort(\mathcal{P}_X)$
2: $P \leftarrow pop(\mathcal{P}_X)$
3: **for all** $P' \in \mathcal{P}_X$ **do**
4: $\quad CG_P \leftarrow graph(P)$
5: $\quad CG_{P'} \leftarrow graph(P')$
6: $\quad l \leftarrow getLink(QG_q, CG_P, CG_{P'})$
7: $\quad P \leftarrow addBinOp(P, P', \bowtie, feat(l))$
8: **end for**
9: **return** P

Algorithm 2 gevJoinToNra($\mathcal{P}_X$, QG_q

First of all, Algorithm 2 details the translation of a GEV join $\hat{\bowtie}$ (X). The plans $\mathcal{P}_X$ corresponding to the GEV creations are sorted based on a minimum selectivity heuristics (Line 1). Based on such ordering, a left-deep tree of join operations is defined (Lines 2 to 8); here, the $getLink()$ function (Line 6) returns the link in QG_q in the middle between CG_P and $CG_{P'}$, so as to identify the feature to use in the join predicate between the results of P and P'.

Remarkably, left-deep trees offer limited inter-operator parallelism capabilities than other forms of trees (`?DBLP:reference/db/2018`). However, the choice of focusing on left-deep trees only is supported by the following considerations. On the one hand, inter-operator parallelism is often limited by several factors, including the larger data footprint required (which may oversubscribe memory bandwidth or induce more spills to disk, i.e., resources that are often under pressure in big data scenarios), the brevity of relational pipelines (which is typical of GPSJ queries), and the presence of blocking operators and/or dependencies between operators (`?DBLP:reference/db/2018`). On the other hand, left-deep trees offer a smaller search space (Dong & Liang, 2007), exploit the cost-reducing pipelining technique on each join operation (Steinbrunn et al., 1997), and allow us to maintain compatibility with our previous work where left-deep trees had been used (Forresi et al., 2021; Forresi et al., 2021).

Require: $\mathcal{P}_X$: a set of NRA plans to compute the LEVs X; p_X: the set of selection predicates to apply to merged LEVs.
Ensure: P: the NRA plan that creates the from X.
1: $\mathcal{P}_X = sort(\mathcal{P}_X)$
2: $P \leftarrow pop(\mathcal{P}_X)$
3: $CG_P = graph(P)$
4: **for all** $P' \in \mathcal{P}_X$ **do**
5: $\quad P \leftarrow addBinOp(P, P', \sqcup, key(gran(CG_P)))$
6: **end for**
7: **if** $p_X \neq \varnothing$ **then**
8: $\quad P_\chi \leftarrow addUnOp(P_\chi, \sigma, \bigwedge_{p \in p_X})$
▷ Add optional selection
9: **end if**
10: **while** $fact(CG_P) \neq gran(CG_P)$ **do**
11: $\quad E \leftarrow E \in \mathcal{E}_P : gran(CG_P) \rightarrow E$
12: $\quad P \leftarrow addUnOp(P, \mu, name(E))$
13: $\quad CG_p = graph(P)$
14: **end while**
15: **return** P

Algorithm 3 gevCreationToNra($\mathcal{P}_X$, p_X)

Algorithm 3 details the translation of a GEV creation $\hat{\sqcup}$ (X, p_X). Similarily to Algorithm 2, the plans $\mathcal{P}_X$ corresponding to the LEV creations are sorted based on a minimum selectivity heuristics (Line 1) and, based on such ordering, a left-deep tree of merge operations is defined (Lines 2 to 6). At Line 8, the plan is possibly extended with the selection operation, if specified. Finally, a sequence of unnest operations is possibly added to return the result in FlR, independently of its original representation Lines 10 to 14).

Algorithm 4 details the translation of a LEV creation operation $\hat{\pi}(\mathcal{C}_\chi, CG_\chi, F_\chi, p_\chi)$. As a LEV may be extracted from several collections, the process is divided into steps: a LEV is first defined for each collection $C \in \mathcal{C}_\chi$ (Lines 3 to 8), then they are progressively put together to build the final LEV (Lines 9 to 22). Depending on $rep(CG_\chi)$, the process follows different strategies. In case of FlR or NoR, the collections are initially sorted (Line 2) based on a minimum selectivity heuristics, and the single-collection LEVs are put together via join operations. In case of NeR, the collections are initially sorted from the finest to the coarsest, and the joins between LEVs are followed by nest operations to produce the nested structure (Lines 16 to 21).

The creation of single-collection LEVs is detailed in Algorithm 5. At a high level, the algorithm is composed of three steps: (i) application of selection predicates, (ii) projection of relevant features, and (iii) remodeling of the data to adapt the result to $rep(CG_\chi)$.

Besides accessing the collection C (Line 2), the first step immediately applies selection predicates to reduce the number of records involved in the following operations

Require: $\mathcal{C}_\chi \subseteq \mathcal{C}$: a set of collections stored in the same database; $CG_\chi = (\mathcal{E}_\chi, L_\chi)$: the collection graph of χ; F_χ the set of features to project; p_χ the set of selection predicates to apply.
Ensure: P_χ: the NRA plan that extracts CG_χ from $\mathcal{C}_\chi$ in the pattern $rep(CG_\chi)$.
1: $P_\chi \leftarrow$ **new** $PlanNode()$
2: $\mathcal{C}_\chi \leftarrow sort(\mathcal{C}_\chi, CG_\chi, p_\chi)$
3: **for all** $C \in \mathcal{C}_\chi$ **do**
4: $\quad \mathcal{E}_{(C,\chi)} \leftarrow \mathcal{E}_C \cap \mathcal{E}_\chi$
5: $\quad F'_\chi \leftarrow F_C \cap F_\chi$
6: $\quad p'_\chi \leftarrow \{p \in p_\chi : feat(p) \in F_C\}$
7: $\quad CG'_\chi \leftarrow subgraph(CG_\chi, \mathcal{E}_{(C,\chi)})$
8: $\quad P_C \leftarrow createSingleCollLEV(C, CG'_\chi, F'_\chi, p'_\chi)$
9: $\quad$ **if** $isEmpty(P_\chi) = true$ **then**
10: $\quad\quad P_\chi \leftarrow P_C$
11: $\quad$ **else**
12: $\quad\quad CG_{P_\chi} = (\mathcal{E}_{P_\chi}, L_{P_\chi}) \leftarrow graph(P_\chi)$
13: $\quad\quad l \leftarrow getLink(CG_\chi, \mathcal{E}_{P_\chi}, \mathcal{E}_{(C,\chi)})$
14: $\quad\quad P_\chi \leftarrow addBinOp(P_\chi, P_C, \bowtie, feat(l))$ ▷ Join them
15: $\quad\quad CG_{P_\chi} \leftarrow graph(P_\chi)$
16: $\quad\quad$ **if** $rep(CG_\chi) =$ NeR **then** ▷ Nest results if NeR is required
17: $\quad\quad\quad F' \leftarrow (\bigcup_{E \in \mathcal{E}'_C : E \Rightarrow gran(C')} F_E) \cap F_\chi$
18: $\quad\quad\quad F'' \leftarrow F_{gran(C')} \cap F_\chi$
19: $\quad\quad\quad predicate \leftarrow (F', name(gran(CG_{P_\chi})), F'')$
20: $\quad\quad\quad P_\chi \leftarrow addUnOp(P_\chi, \nu, predicate)$
21: $\quad\quad$ **end if**
22: $\quad$ **end if**
23: **end for**
24: **return** P_χ

Algorithm 4 levCreationToNra($\mathcal{C}_\chi, CG_\chi, F_\chi, p_\chi$)

(Lines3 to 9). For each feature that needs a selection, we build a disjunction of predicates that consider every schema variation of such feature.

The second step takes into consideration schema variation and column pruning. Its goal is to project only the relevant attributes and to map them to features (by applying the respective transcoding functions and renaming them to the feature names, Lines 11 to 18). In particular, for each feature $f \in F_\chi$ representing attributes in $\mathcal{S}_C$ we project a single attribute (named after $rep(f)$) that contains the only non-null value among its schema variations (simplified in Line 15 as a disjunction over each $a \in A$).

The third step consists of the NRA operations to adapt the obtained collection to the representation specified in input, i.e., $rep(CG_\chi)$ (Lines 20 to 26). Three kinds of operations are possibly applied.

- Unnest operations (Lines 20 to 26) are applied to flatten collection records, thus moving the granularity from a coarse entity to a finer one. For instance, this operation is necessary to obtain C_j from C_i in Fig. 7.
- An aggregation operation (Lines 27 to 32) is applied to coarsen the aggregation level of a collection. For instance, this operation is necessary to keep only customer and order records (i.e., E_{or} and E_{cu}) from C_k in Fig. 7, which results in "moving" the fact from E_{ol} to E_{or}.
- Nest operations (Lines 33 to 43) are applied to nest collection records, thus moving the granularity from a fine entity to a coarser one. For instance, this is necessary to obtain C_i from C_k in Fig. 7.

Require: C: a collection; CG_χ: the collection graph of χ; F_χ the set of features to project; p_χ the set of selection predicates to apply.
Ensure: P_χ: the NRA plan that extracts CG_χ from C.
1: $P_\chi \leftarrow$ **new** $PlanNode()$
2: $P_\chi \leftarrow addUnOp(P_\chi, \mathsf{CA}, C)$ ▷ Access the collection
3: $ps \leftarrow \varnothing$ ▷ Predicate set
4: **for all** $s \in p_\chi$ **do**
5: $\quad A \leftarrow \mathcal{S}_C \cap f$
6: $\quad ps \leftarrow ps \cup (\bigvee_{a \in A}(\varphi_{(rep(feat(s),a)}(a), \omega(s), \upsilon(s))))$
7: **end for**
8: **if** $ps \neq \varnothing$ **then**
9: $\quad P_\chi \leftarrow addUnOp(P_\chi, \sigma, \bigwedge_{p \in ps})$ ▷ Optional selections
10: **end if**
11: $ps \leftarrow \emptyset$
12: **for all** $f \in F_\chi$ **do**
13: $\quad A \leftarrow \mathcal{S}_C \cap attr(f)$
14: $\quad$ **if** $A \neq \varnothing$ **then**
15: $\quad\quad ps \leftarrow (\bigvee_{a \in A} \varphi_{(rep(f),a)}(a)) \,/\, rep(f)$
16: $\quad$ **end if**
17: **end for**
18: $P_\chi \leftarrow addUnOp(P_\chi, \pi, ps)$ ▷ Projection of features
19: $CG'_C \leftarrow graph(P_\chi)$ ▷ Projections could change $fact(CG_C)$
20: **while** $gran(C') \neq fact(CG'_C) \wedge gran(C') \not\Rightarrow gran(\chi)$ **do**
21: $\quad E \leftarrow E \in \mathcal{E}'_C : gran(C') \rightarrow E$
22: $\quad$ **for all** $a \in arr(C, E)$ **do**
23: $\quad\quad P_\chi \leftarrow addUnOp(P_\chi, \mu, a)$
24: $\quad$ **end for**
25: $\quad CG'_C \leftarrow graph(P_\chi)$
26: **end while**
27: **if** $fact(CG'_C) \neq fact(CG_\chi)$ **then**
28: $\quad F' \leftarrow (\bigcup_{E \in \mathcal{E}_\chi} F_E) \cap F_\chi$
29: $\quad predicate \leftarrow (F', \varnothing)$
30: $\quad P_\chi \leftarrow addUnOp(P_\chi, \gamma, predicate)$ ▷ Fact coarsening
31: $\quad CG'_C \leftarrow graph(P_\chi)$
32: **end if**
33: **while** $gran(C') \Rightarrow gran(\chi)$ **do**
34: $\quad F' \leftarrow (\bigcup_{E \in \mathcal{E}'_C : E \Rightarrow gran(C')} F_E) \cap F_\chi$
35: $\quad F'' \leftarrow F_{gran(C')} \cap F_\chi$
36: $\quad$ **if** $\exists E \in \mathcal{E}'_C : E \rightarrow gran(C')$ **then**
37: $\quad\quad E \leftarrow E \in \mathcal{E}'_C : E \rightarrow gran(C')$
38: $\quad\quad F'' \leftarrow name(E) \cup F''$
39: $\quad$ **end if**
40: $\quad predicate \leftarrow (F', name(gran(C')), F'')$
41: $\quad P_\chi \leftarrow addUnOp(P_\chi, \upsilon, predicate)$ ▷ Granularity coarsening
42: $\quad CG'_C \leftarrow graph(P_\chi)$
43: **end while**
44: **return** P_χ

Algorithm 5 $createSingleCollLEV(C, CG_\chi, F_\chi, p_\chi)$

Funding Open access funding provided by Alma Mater Studiorum - Università di Bologna within the CRUI-CARE Agreement.

Declarations

Conflict of Interests All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

References

Agrawal, D., Chawla, S., Contreras-Rojas, B., & et al (2018). RHEEM: enabling cross-platform data processing - may the big data be with you! -. *Proceedings of the VLDB Endowment*, *11*(11), 1414–1427. https://doi.org/10.14778/3236187.3236195.

Atzeni, P., Bugiotti, F., & Rossi, L (2014). Uniform access to nosql systems. *Information Systems*, *43*, 117–133. https://doi.org/10.1016/j.is.2013.05.002.

Baldacci, L., Golfarelli, M., Lombardi, D., et al. (2016). Natural gas consumption forecasting for anomaly detection. *Expert Systems with Applications*, *62*, 190–201. https://doi.org/10.1016/j.eswa.2016.06.013.

Ben Hamadou, H., Gallinucci, E., & Golfarelli, M (2019). Answering GPSJ queries in a polystore: a dataspace-based approach. In *Proceedings of conceptual modeling - 38th int. conf., ER 2019*, (Vol. 11788 pp. 189–203). Springer. https://doi.org/10.1007/978-3-030-33223-5_16.

Bimonte, S., Gallinucci, E., Marcel, P., & et al (2021). Data variety, come as you are in multi-model data warehouses. Information Systems. https://doi.org/10.1016/j.is.2021.101734.

Bleiholder, J., & Naumann, F. (2005). Declarative data fusion - syntax, semantics, and implementation. In *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. https://doi.org/10.1007/11547686_5.

Bleiholder, J., & Naumann, F. (2008). Data fusion. *ACM Computing Surveys*, *41*(1), 1,1–1,41. https://doi.org/10.1145/1456650.1456651.

Bonaque, R. et al. (2016). Mixed-instance querying: a lightweight integration architecture for data journalism. *Proceedings of the VLDB Endowment*, *9*(13), 1513–1516. https://doi.org/10.14778/3007263.3007297.

Bondiombouy, C., & Valduriez, P. (2016). Query processing in multistore systems: an overview. *International Journal of Cloud Computing*, *5*(4), 309–346. https://doi.org/10.1504/IJCC.2016.10001884.

Darmont, J., Boussaid, O., & Bentayeb, F (2005). DWEB: a data warehouse engineering benchmark. In A.M. Tjoa, & J. Trujillo (Eds.) *Data warehousing and knowledge discovery, 7th international conference, DaWaK 2005, Copenhagen, Denmark, August 22-26, 2005, proceedings, lecture notes in computer science*, (Vol. 3589 pp. 85–94). Springer. https://doi.org/10.1007/11546849_9.

Delaney, I., & Li, J. (2017). Extending apache spark sql data source apis with join push down. https://databricks.com/session/extending-apache-spark-sql-data-source-apis-with-join-push-down, [Online; accessed 10-Sep-2021].

Deshpande, A., & Hellerstein, J.M. (2002). Decoupled query optimization for federated database systems. In R. Agrawal, & K.R. Dittrich (Eds.) *Proceedings of the 18th international conference on data engineering, San Jose, CA, USA, February 26 - March 1, 2002* (pp. 716–727). IEEE Computer Society. https://doi.org/10.1109/ICDE.2002.994788.

DiScala, M., & Abadi, D.J. (2016). 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. https://doi.org/10.1145/2882903.2882924.

Dong, H., & Liang, Y. (2007). Genetic algorithms for large join query optimization. In H. Lipson (Ed.) *Genetic and evolutionary computation conference, GECCO proceedings, London, England, UK, July 7-11, 2007* (pp. 1211–1218). ACM. https://doi.org/10.1145/1276958.1277193.

Duggan, J., Elmore, A.J., Stonebraker, M., & et al (2015). The bigdawg polystore system. *SIGMOD Record*, *44*(2), 11–16. https://doi.org/10.1145/2814710.2814713.

Forresi, C., Francia, M., Gallinucci, E., & et al (2021). Optimizing execution plans in a multistore. In *Advances in databases and information systems - 25th European conference, ADBIS 2021, Tartu, Estonia, August 24-26, 2021, Proceedings* (pp. 136–151). Springer. https://doi.org/10.1007/978-3-030-82472-3_11.

Forresi, C., Gallinucci, E., Golfarelli, M., & et al (2021). A dataspace-based framework for olap analyses in a high-variety multistore. *The VLDB Journal*, 1–24. https://doi.org/10.1007/s00778-021-00682-5.

Francia, M., Gallinucci, E., Golfarelli, M., & et al (2021). Making data platforms smarter with MOSES. *Future Generation Computer Systems*, *125*, 299–313. https://doi.org/10.1007/s00778-021-00682-5.

Franklin, M.J., Halevy, A.Y., & Maier, D (2005). From databases to dataspaces: a new abstraction for information management. *SIGMOD Record*, *34*(4), 27–33. https://doi.org/10.1007/s00778-021-00682-5.

Gadepally, V., Chen, P., Duggan, J., & et al (2016). The bigdawg polystore system and architecture. In *2016 IEEE High performance extreme computing conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016* (pp. 1–6). https://doi.org/10.1109/HPEC.2016.7761636.

Gallinucci, E., Golfarelli, M., & Rizzi, S (2019). Approximate OLAP of document-oriented databases: a variety-aware approach. *Information Systems*, *85*, 114–130. https://doi.org/10.1016/j.is.2019.02.004.

Gog, I., Schwarzkopf, M., Crooks, N., & et al (2015). Musketeer: all for one, one for all in data processing systems. In L. Réveillère, T. Harris, & M. Herlihy (Eds.) *Proceedings of the tenth European conference on computer systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015* (pp. 1–16). ACM. https://doi.org/10.1145/2741948.2741968.

Golfarelli, M., Graziani, S., & Rizzi, S (2019). An active learning approach to build adaptive cost models for web services. *Data and Knowledge Engineering*, *119*, 89–104. https://doi.org/10.1016/j.datak.2019.01.001.

Golfarelli, M., Maio, D., & Rizzi, S (1998). The dimensional fact model: a conceptual model for data warehouses. *International Journal of Cooperative Information Systems*, *7*(2-3), 215–247. https://doi.org/10.1142/S0218843098000118.

Golfarelli, M., & Rizzi, S. (2009). *Data warehouse design: modern principles and methodologies*. McGraw-Hill, Inc. https://doi.org/10.5555/1594749.

Golfarelli, M., & Saltarelli, E. (2003). The workload you have, the workload you would like. In S. Rizzi, & I. Song (Eds.) *DOLAP 2003, ACM Sixth international workshop on data warehousing and OLAP, New Orleans, Louisiana, USA, November 7 Proceedings* (pp. 79–85). ACM. https://doi.org/10.1145/956060.956075.

Jeffery, S.R., Franklin, M.J., & Halevy, AY (2008). Pay-as-you-go user feedback for dataspace systems. In *2008 ACM SIGMOD Int. conf. on management of data* (pp. 847–860). ACM. https://doi.org/10.1007/978-3-319-13704-9_7.

Kaitoua, A., Rabl, T., Katsifodimos, A., & et al (2019). Muses: distributed data migration system for polystores. In *35th IEEE international conference on data engineering, ICDE 2019, Macao, China, April 8-11, 2019* (pp. 1602–1605). IEEE. https://doi.org/10.1109/ICDE.2019.00152.

Kolev, B. et al. (2016). Cloudmdsql: querying heterogeneous cloud data stores with a common language. *Distributed and Parallel Databases*, *34*(4), 463–503. https://doi.org/10.1007/s10619-015-7185-y.

Liu, L., & Özsu, M.T. (Eds.) (2018). Encyclopedia of database systems, 2nd edn. New York: Springer. https://doi.org/10.1007/978-1-4614-8265-9.

Loader, C. (2006). *Local regression and likelihood*. Springer Science & Business Media. https://doi.org/10.1007/b98858.

Lu, J., & Holubová, I. (2019). Multi-model databases: a new journey to handle the variety of data. *ACM Computing Surveys*, *52*(3), 55,1–55, 38. https://doi.org/10.1145/3323214.

Maccioni, A., & Torlone, R. (2018). Augmented access for querying and exploring a polystore. In *34th IEEE Int. conf. on data engineering, ICDE 2018* (pp. 77–88). IEEE Computer Society. https://doi.org/10.1109/ICDE.2018.00017.

Mandreoli, F., & Montangero, M. (2019). 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. https://doi.org/10.1016/B978-0-444-63984-4.00009-0.

Mazumdar, S., Seybold, D., Kritikos, K., et al. (2019). A survey on data storage and placement methodologies for cloud-big data ecosystem. *Journal of Big Data*, *6*(1), 15. https://doi.org/10.1186/s40537-019-0178-3.

O'Neil, P.E., O'Neil, E.J., Chen, X., & et al (2009). The star schema benchmark and augmented fact table indexing. In R.O. Nambiar, & M. Poess (Eds.) *Performance evaluation and benchmarking, first TPC technology conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers, Lecture Notes in Computer Science*, (Vol. 5895 pp. 237–252). Springer. https://doi.org/10.1007/978-3-642-10424-4_17.

Rafique, A., Van Landuyt, D., Reniers, V., & et al (2017). Towards an adaptive middleware for efficient multi-cloud data storage. In *Proceedings of the 4th workshop on crosscloud infrastructures & platforms* (pp. 1–6). https://doi.org/10.1145/3069383.3069387.

Sadalage, P.J., & Fowler, M. (2013). NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Pearson Education. https://doi.org/10.5555/2381014.

Sellami, R., & Defude, B. (2018). Complex queries optimization and evaluation over relational and nosql data stores in cloud environments. *IEEE Transactions on Big Data*, *4*(2), 217–230. https://doi.org/10.1109/TBDATA.2017.2719054.

Singhal, R., Zhang, N., Nardi, L., & et al (2019). Polystore++: accelerated polystore system for heterogeneous workloads. In *39th IEEE International conference on distributed computing systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019* (pp. 1641–1651). IEEE. https://doi.org/10.1109/ICDCS.2019.00163.

Steinbrunn, M., Moerkotte, G., & Kemper, A (1997). Heuristic and randomized optimization for the join ordering problem. *VLDB Journal*, *6*(3), 191–208. https://doi.org/10.1007/s007780050040.

Subramanian, D.K., & Subramanian, K. (1998). Query optimization in multidatabase systems. *Distributed Parallel Databases*, *6*(2), 183–210. https://doi.org/10.1023/A:1008691331104.

Tan, R., Chirkova, R., Gadepally, V., et al. (2017). Enabling query processing across heterogeneous data models: a survey. In *2017 IEEE Int. conf. on big data* (pp. 3211–3220). IEEE Computer Society. https://doi.org/10.1109/BigData.2017.8258302.

The myria big data management and analytics system and cloud services (2017)

Zhang, C., Lu, J., Xu, P., & et al (2018). Unibench: a benchmark for multi-model database management systems. In *Performance evaluation and benchmarking for the era of artificial intelligence - 10th TPC technology conference, TPCTC 2018*, (Vol. 11135 pp. 7–23). Springer. https://doi.org/10.1007/978-3-030-11404-6_2.

Chiara Forresi graduated cum laude in Computer Science and Engineering in October 2020 at the University of Bologna. She is currently a Ph.D. student in Data Science and Computation. Her main research interests are about techniques and methodologies to support data management and analysis in big data ecosystems.

Matteo Francia received his Ph.D. in Computer Science and Engineering from the University of Bologna, Italy. He is an adjunct professor and a post-doc research fellow at the University of Bologna. His research focuses on advanced analytics and unconventional data, with particular reference to IoT and spatio-temporal data.

Enrico Gallinucci is adjunct professor at the University of Bologna, where he received his Ph.D. in Computer Science and Engineering and teaches Business Intelligence and Big Data. His research interests currently focus on big data analytics, NoSQL and multimodel database systems, data democratization, and precision agriculture. He is associate editor for the DKE journal.

Matteo Golfarelli is full professor at the University of Bologna. He is author of over 130 publications in international journals and conferences mainly in the areas of database systems and business intelligence. His current research interests include Big Data Analytics, Machine Learning, NoSQL. He is member of the steering committee of DOLAP and associate editor for DKE and Electronics journals. He is the coordinator of the International Master Degree in Digital Transformation Management.