Augmented Access for Querying and Exploring a Polystore

Antonio Maccioni
Collective[i], New York City, USA
amaccioni@collectivei.com

Riccardo Torlone
Roma Tre University, Rome, Italy
torlone@dia.uniroma3.it

***Abstract*—The huge diversity of database technologies in use inside organizations pose today new challenges of data management and integration. Polystores provide a solution to this scenario based on a loosely coupled integration of data sources and the direct access, with the local language, to each storage engine for exploiting its distinctive features. However, given the absence of a global schema, it is hard to know if a query to one system can be satisfied with data stored elsewhere in the polystore. We address this issue by introducing *query augmentation*, a data manipulation operator for polystores based on the automatic enrichment of the answer to a local query with related data in the rest of the polystore. Augmentation can be used to implement two effective methods for data access in polystores: *augmented search* and *augmented exploration*. We show that they provide effective tools for information discovery in polystores that avoid middleware layers, abstract query languages, and shared data models. We also illustrate the design of QUEPA, a system that fully implements our approach in an efficient way. A comprehensive campaign of experiments done with QUEPA shows that our approach is feasible and, unlike other approaches, scales nicely as the polystore grows in the number of stores and size of databases.**

I. Introduction

When dealing with data management, organizations are becoming polyglot: they tend to use the most suitable database system depending on the specific kind of data they manage and/or the specific activity [12], [17], [18], [23], [30]. Recent research has shown that, on average, each enterprise application is backed by at least two or three different types of database engines [32].

Running Example 1: Let us consider, as a practical example, the databases of a company called *Polyphony* selling music online. As shown in Fig. 1, each department uses a storage system that best fits its specific business objectives: (i) the sales department guarantees ACID properties for its transactions database with a relational system, (ii) a warehouse department supports search operations with a document store catalogue, where each item is represented by a JSON document, and (iii) a marketing department uses a graph database of similar-items supporting recommendations. In addition, there exists a key-value store containing discounts on products, which is shared among the three departments above.

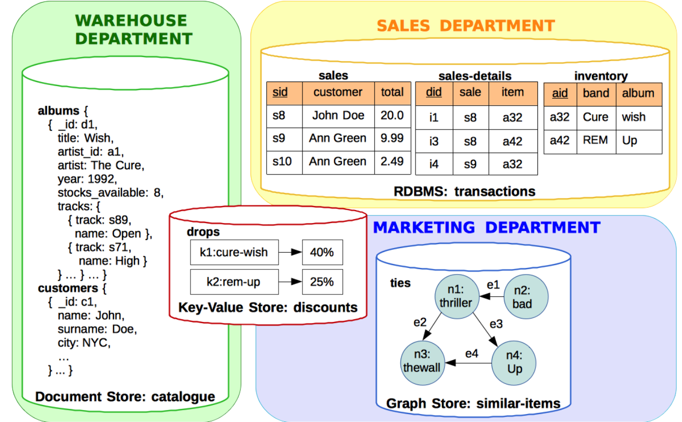


Fig. 1. A polyglot environment.

In this framework, it is of strategic importance to provide mechanisms for searching through all the available data, possibly by means of a simple user interface [14]. The traditional approach to address this issue is based on a middleware layer involving a unified language, a common interface, or a universal data model [3], [6], [17], [20], [33], [34]. However, this solution adds computational overhead at runtime and, more importantly, hides the specificity and functionality that these systems were adopted for [30]. In addition, it is hard to maintain, having an inherent complexity that increases significantly as new database systems take part to the environment.

Polystore systems (or simply, polystores) have been proposed recently as an alternative solution for this scenario [30]. The basic idea is to provide a loosely coupled integration of data sources and allow direct access, with the local language, to each specific storage engine to exploit its distinctive features. This approach meets the “one size does not fit all” philosophy as well as the need to support business cases where heterogeneous databases have to co-exist.

In polystores, it is common that a user is only aware of a single (or a few) available database but does not know anything about other databases (neither the content, nor the way to query them and, sometimes, not even their existence). This clearly poses new challenges for accessing and integrating data in polystores in an effective way. To recall a relevant discussion about this approach, the issue is that *“if I knew what query to ask, I would ask it, but I don’t”* [30].

In this paper, we provide a contribution to this problem by proposing, in a formal way, (*query*) *augmentation*, a new construct for data manipulation in polystores that, based on the simple notion of probabilistic relationship between objects in

different data stores, allows the enrichment of query answering over a local database with data outside the database but available in the polystore. The implementation of this operator does not require the addition of an abstraction layer involving query translation and therefore has a minimal impact on the applications running on top of the data layer. The goal is to provide a soft mechanism for data integration in polystores that complements other approaches, such as those based on cross-db joins [3], [12], [17], [20].

Two effective methods for data access in polystore can be defined with the augmentation construct: *augmented search* and *augmented exploration*.

Augmented search consists of the automatic expansion of the result of a query over a local database with data that is relevant to the query but which is stored elsewhere in the polystore. This is very useful in common scenarios where information is shared across the organization and the various databases complement or overlap each other. Assume for instance that Lucy, an employee of Polyphony working in the sales department who only knows SQL, needs all the information available on the album "Wish". Then, she submits in *augmented* mode the following query to the relational database transactions in Fig. 1.

```
SELECT *
FROM inventory
WHERE name like '%wish%'
```

By exploiting augmentation, the result of this query is the augmented object reported below, revealing details about the product that are not in the database of the sales department, including the fact that it is currently on a 40% discount.

< a32, Cure, Wish > ⇒ (discounts: 40%)
⇓
(catalogue:{ title: Wish,
artist_id: a1,
artist: The Cure,
year: 1992,
... })

In an augumented search, each retrieved element e is associated with the probability that e is related to an element of the original result. Such probability is derived off-line from mining techniques and integrity constraints. Colors (as in the example above) and rankings can be used in practice to represent probability in a more intuitive way.

Augmented exploration makes use of the augmentation operator to provide a more interactive and flexible way to access data, which consists of a guided expansion of the result of a local query. For example, if Lucy submits in *exploratory* mode another SQL query to the database transactions for retrieving all the sales whose total income is greater than 15, she obtains the tuple that follows, in which the links suggest that further information, related to the returned tuple, is available elsewhere and allows her to decide with a click how to deepen the result. As above, the result is probabilistic.

<s8, John Doe, 20.0 > --on-click--→ ({ _id: c1,
name: John,
surname: Doe,
city: NYC,
... })

This process is iterative and provides a method of database exploration [7], [9], [29], where the user can freely find her way through the polystore, by just clicking on the links as soon as they are made available.

We have implemented our approach in QUEPA[1], a polystore system equipped with the augmentation operator that provides the access methods discussed above and is compatible with most modern database engines. QUEPA operates in a plug-and-play mode and does not affect the access modalities of the various storage systems, thus reducing the need for ad-hoc configurations and for a middleware involving unified query languages or shared data models. In addition, we have developed an optimization technique that relies on machine learning methods to tune the execution of the augmentation.

In sum, the contributions of this paper are the following:

- We propose a general data model for polystores and introduce formally, in this model, an operator for data manipulation in polystores that: (i) provides a lightweight mechanism for data integration, (ii) keeps data in the original format, (iii) allows the use of the original query languages or APIs and (iv) avoids any query translation.
- We present two effective methods for data access in polystores that rely on the new operator: augmented search enhances the capability of standard query answering in polystores while augmented exploration adds to the picture a mechanism for database exploration.
- We illustrate in detail a series of techniques for efficiently using memory, CPU and network resources available in the polystore system when augmenting a query. In addition, we describe a rule-based optimizer that relies on machine learning for deciding among different options for executing the augmentation. In polystores, this technique is easier to employ than cost-based optimizers, usually adopted by the state-of-the-art.
- We discuss a comprehensive campaign of experiments on a polystore involving several popular database systems. The results demonstrate that the approach is feasible and that the adopted optimization techniques guarantee an efficient execution of the augmentation operator, also when compared with other approaches.

The rest of the paper is organized as follows. In Section II we introduce the augmentation operator and the two access methods based on it. In Section III, we describe the architecture of a system supporting augmentation, while in Section IV we illustrate the techniques for executing the augmentation more efficiently. In Section V we discuss our rule-based optimizer and, in Section VI, we compare our approach with related work. Section VII contains the experimental results and Section VIII our conclusions.

[1]A demo of the first release of QUEPA has been shown in [18].

II. Augmented Access to Polystores

A. A general data model for polystores

In PDM (Polystore Data Model) a polystore $\mathcal{P}$ is made of a set of databases $\mathcal{P} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ stored in a variety of data management systems $\mathcal{S}_1, \ldots, \mathcal{S}_n$ respectively (relational, key-value, graph, etc.). A database $\mathcal{D} \in \mathcal{P}$ consists of a set of *data collections* $\mathcal{D} = \{C_1, \ldots, C_k\}$ where each data collection C is a set of *(data) objects*. An object $o \in C$ is just a key-value pair: $o = \langle k, v \rangle$ where k identifies uniquely o in C and v is an atomic piece of data. A tuple and a JSON document are examples of data objects in a relational database and in a document store of a polystore, respectively.

By definition, given a database $\mathcal{D}$ of a polystore $\mathcal{P}$, a data collection C in $\mathcal{D}$ and a data object $o = \langle k, v \rangle$ in C, we can uniquely identify o in $\mathcal{P}$ by means of k, C and $\mathcal{D}$. We call $\hat{k} = \mathcal{D}.C.k$ the *global-key* of o in $\mathcal{P}$.

Example 1: Consider the polystore scenario in Fig. 1. Document $\langle \mathsf{d1}, \mathsf{_id : d1}, \mathsf{title : Wish}, \ldots \rangle$ is an object in the (unique) collection of the catalogue database whereas tuple $\langle \mathsf{s8}, (\mathsf{s8}, \mathsf{John\ Doe}, 20.0) \rangle$ is an object of the collection sales in the transactions database. The global-key of the latter is transactions.sales.s8.

This model captures polystores involving any database system satisfying the minimum requirement that every stored data object can be identified and accessed by means of a key. It should be noted however that the granularity of objects inevitably depends on the designer's choice (see for instance the discussion in [5]) and the nature of data (less structured databases are expected to have more coarse-grained objects).

The other main ingredient of PDM is the ability to correlate data objects of possibly different databases of the polystore by means of the following basic notion, which we call *p-relation* (for relation in a polystore).

Definition 1 (p-relation): A p-relation on two objects o_1 and o_2, denoted by $o_1 R_p o_2$, represents the existence of a relation R between o_1 and o_2 with probability p $(0 < p \leq 1)$, where R can be one of the following types:

- the *identity*, denoted by $\sim$: a reflexive, symmetric and transitive relation (i.e., an equivalence relation), representing the fact that o_1 and o_2 refer to the same real-world entity;
- the *matching*, denoted by $\rightleftharpoons$: a reflexive and symmetric relation (not necessarily transitive), representing the fact that o_1 and o_2 share some common information.

We assume that in a polystore, p-relations are "consistent" in the sense that they always satisfy the following condition.

Consistency Condition. For each triple of objects o_1, o_2 and o_3 in a polystore $\mathcal{P}$ such that $o_1 \rightleftharpoons o_2$ and $o_2 \sim o_3$ it is also the case that $o_1 \rightleftharpoons o_3$.

While this is a natural condition (two equivalent objects should match the same objects), it guarantees that the augmentation construct behaves consistently with equivalent objects, as we will show in the following.

Example 2: Consider the polystore in Fig. 1. By denoting the objects with their global keys we have for instance that:

- catalogue.albums.d1 $\sim_{0.8}$ discount.drop.k1:cure:wish,
- catalogue.albums.d1 $\sim_{0.9}$ transactions.inventory.a32,
- transactions.inventory.a42 $\sim_{0.6}$ similarItems.ties.n4,
- transactions.inventory.a32$\rightleftharpoons_1$transactions.sales-details.i4.

Basically, while the identity relation serves to represent multiple occurrences of the same entity in the polystore, the matching relation models general relationships between data different from the identity (e.g., those typically captured by foreign keys in relational databases or by links in graph databases). On the practical side, p-relations are derived from the metadata associated with databases in the polystore (e.g., from integrity constraints) or are discovered using probabilistic mining techniques. For the latter task, we rely on the state-of-the-art techniques for probabilistic record linkage [22], that is, algorithms able to score the likelihood that a pair of objects in different databases match. Note however that also deterministic techniques can be used here by assigning $p = 1.0$ to each retrieved link. In Section III-D, we will give more details on methods and tools currently used in the implementation of our approach for collecting and score p-relations.

B. The augmentation construct

This section gives a more formal definition of the augmentation construct, which is the basic operator of our approach. It takes as input an object o of a polystore and returns the augmented set $\alpha^n(o)$, which iteratively returns data objects in the polystore that are related to o with a certain probability. This probability is computed by combining the probabilities of the relationships that connect o with the retrieved objects.

Definition 2 (Augmentation construct): Let $\mathbf{o}$ be a set of objects in a polystore $\mathcal{P}$. The *augmentation* α^n *of level* $n \geq 0$ of $\mathbf{o}$ is a set $\mathbf{o}'$ of objects o^p, where $o \in \mathcal{P}$ and p is the probability of membership of o to $\mathbf{o}'$, defined as follows $(m > 0)$:

- $\alpha^0(\mathbf{o}) = \mathbf{o} \cup \{o^p \mid o \sim_p o' \wedge o' \in \mathbf{o}\}$
- $\alpha^m(\mathbf{o}) = \alpha^{m-1}(\mathbf{o}) \cup \{o^{\hat{p}} \mid o \rightleftharpoons_{p'} o' \wedge o'^p \in \mathbf{o} \wedge \hat{p} = p \cdot p'\}$

Example 3: Let o be the object in the polystore in Fig. 1 with global-key catalogue.albums.d1. Then, according to the p-relations in Example 2 we have $\alpha^0(\{o\}) = \{o, o_1^{0.8}, o_2^{0.9}\}$ where o_1 and o_2 are the objects with global-key discount.drop.k1:cure:wish and transactions.inventory.a32 respectively.

Note that, by combining probabilities by multiplication, we are assuming strong independence between each p-relation. We believe that this is reasonable in a highly heterogeneous framework in which no assumption can be made on the relationships between objects belonging to different databases.

Note also that, by construction, for every $k \geq 0$, if $o \in \alpha^k(\mathbf{o})$ then $o' \in \alpha^k(\mathbf{o})$ for each $o' \sim o$. Indeed, if o belongs to $\alpha^k(\mathbf{o})$ because of an object $o'' \in \alpha^{k-1}(\mathbf{o})$ such that $o'' \rightleftharpoons o$, the consistency condition above implies that if $o \sim o'$, we have also that $o'' \rightleftharpoons o'$ and so, by Definition 2, $o' \in \alpha^k(\mathbf{o})$.

The augmented construct can be at the basis of two alternative ways to access a polystore. They will be illustrated in the following two subsections.

C. Augmented Search

An augmented search consists of the expansion of the result of a query over a local database with data that are relevant to the query but are stored elsewhere in the polystore.

Consider again a polystore $\mathcal{P}$ composed of a set of databases stored in different data management systems each equipped with a specific query language. Now, let $Q^{\mathcal{S}}$ denote a query expressed in the query language of the storage system $\mathcal{S}$ and let $Q^{\mathcal{S}}(\mathcal{D})$ be the set of objects in the result of $Q^{\mathcal{S}}$ over a database $\mathcal{D}$ stored in $\mathcal{S}$.

Definition 3 (Augmented search): The *augmentation of level* $n \geq 0$ of a query $Q^{\mathcal{S}}$ over a database $\mathcal{D}$ stored in $\mathcal{S}$, denoted by $Q^{\mathcal{S}}(n)(\mathcal{D})$, consists in the augmentation of level $n \geq 0$ of the result of $Q^{\mathcal{S}}$ over $\mathcal{D}$ ordered according to the probability of its elements.

Example 4: Let Q be an SQL query over the relational database **transactions** in Fig. 1 that returns the object o with global-key catalogue.albums.d1. Then we have $Q(0)(\textbf{transactions}) = (o, o_2^{0.9}, o_1^{0.8})$, where o_1 and o_2 are the objects with global-key discount.drop.k1:cure:wish and transactions.inventory.a32, and $Q(1)(\textbf{transactions}) = (o, o_2^{0.9}, o_3^{0.9}, o_4^{0.9}, o_1^{0.8})$, where o_3 and o_4 are the objects with global-key transactions.sales−details.i1 and transactions.sales−details.i4.

D. Augmented Exploration

Exploratory computing is a new trend in big data access that aims at helping users make sense of very big data sets by means of step-by-step interaction with the system oriented to the progressive refinement of the data retrieval process [7], [8], [11]. The augmentation construct can provide effective support for exploratory computing in a polystore through a process that we call *augmented exploration*. Intuitively, augmented exploration consists of a guided expansion of the result of a query over a local database with related data stored elsewhere in the polystore. It works as follows: we start with a query $Q^{\mathcal{S}}$ expressed in the query language of the storage system $\mathcal{S}$ in the polystore and execute $Q^{\mathcal{S}}$ over a database $\mathcal{D}$ stored in $\mathcal{S}$. We then select, from the answer of the query an object o and apply to o the augmentation construct of level 0 (step 1) and order the result according to the probability of each element. Again, we select, from the result we obtain, an object o_1 and apply to it the augmentation construct of level 1 (step 2) and order the result. We then proceed similarly, by selecting an object o_i from the result of the previous step and apply the augmentation construct of level 1 to o_i, until the user is satisfied with her search. This can be formalized as follows.

Definition 4 (Augmented exploration): An *augmented exploration* of a polystore $\mathcal{P}$ starting from a query $Q^{\mathcal{S}}$ over a database $\mathcal{D}$ stored in $\mathcal{S}$ consists of a sequence of k steps: $[(o_0 \dashrightarrow \mathbf{o}_0); (o_1 \dashrightarrow \mathbf{o}_1); \ldots; (o_k \dashrightarrow \mathbf{o}_k)]$ where:

- $o_0 \in Q^{\mathcal{S}}(\mathcal{D})$ and $\mathbf{o}_0 = \alpha^0(\{o_0\})$,
- $o_i \in \mathbf{o}_{i-1}$ and $\mathbf{o}_i = \alpha^1(\{o_i\})$ $(i > 0)$.

Example 5: Consider the query in Example 4 and the objects therein discussed. Then, a possible augmented exploration involves the steps:

$$(o \dashrightarrow \{o_1, o_2\}); (o_1 \dashrightarrow \{o_3, o_4\}); (o_3 \dashrightarrow \{o_6\})$$

where o_6 has global-key catalogue.customers.c1.

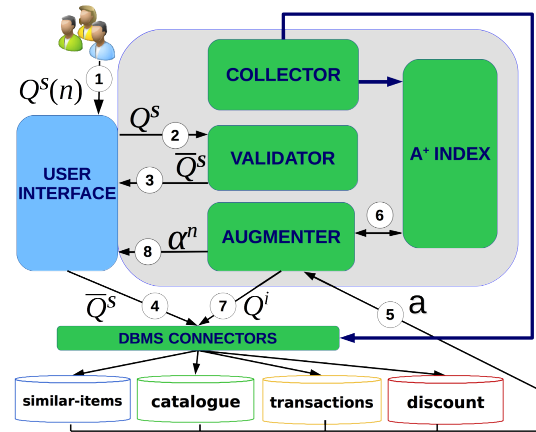


Fig. 2. Architecture of QUEPA.

III. A SYSTEM SUPPORTING AUGMENTATION

A. Architecture of components and their interactions

The architecture of the system is in Fig. 2. The components are briefly described as follows:

- *Augmenter:* implements the augmentation operator and orchestrates augmented query answering. Section IV and Section V will give more details about different variants of this component.
- A+ *index:* stores p-relations and will be described in more detail in Section III-B and Section III-C.
- *Collector:* this component is in charge of discovering, gathering and storing p-relations in the A+*index*. Since this aspect is not a contribution of our work, the Collector component will be briefly described in the next Section III-D.
- *Connectors:* they are used to interact with the polystore. Each connector is able to communicate with a specific database system by sending queries in the local language and returning the result. Data objects are parsed into an internal representation.
- *Validator:* is used to assess whether a query can be augmented or not. For example, queries containing aggregative functions cannot be augmented. The validator can also rewrite queries by adding all identifiers of data objects that are not explicitly mentioned in the query.
- *User Interface:* receives inputs and shows the results using a REST interface.

Since QUEPA does not store any data, it is easy to deploy multiple instances of the system that can answer independent queries in parallel. In this case, each instance has its own A+*index* replica and its own augmenter. Now we show the interactions among the components of QUEPA for answering a query $Q^{\mathcal{S}}$ in augmented mode with level n (step ① in Figure 2).

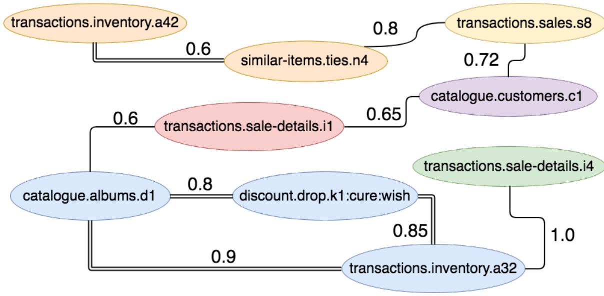


Fig. 3. A+ *index*.

The *validator* first checks if the query is correct (step ②) and possibly rewrites it into $\overline{Q}^S$ (step ③) before its execution over the target database (step ④). The local answer $\mathbf{a}$ is returned to the *augmenter* which is now ready to compute the augmentation (step ⑤). It gets from the A+ *index* the global keys of data objects reachable from $\mathbf{a}$ with n applications of the augmentation primitive (step ⑥). These global keys are used to retrieve data objects from the polystore with local queries Q^i (step ⑦). Finally, the augmented answer is returned to the user (step ⑧). The interaction in the augmented exploration is similar but simplified since, at each step, only a single data object is augmented.

Figure 6(a) illustrates an augmentation process, in which circles stand for data objects and each database is represented by a different color. The original answer contains four results, i.e. the green circles. Each result is connected, by means of arrows, to the objects to include in the augmented answer. The augmentation iterates over the four results and retrieves 11 additional objects with 11 direct-access queries.

B. The A+ Index

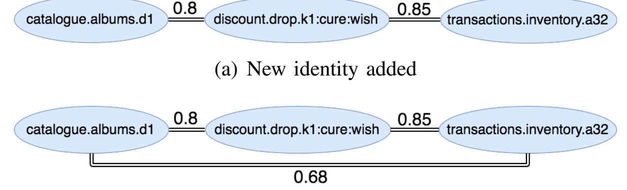
The augmenters use a repository, called A+ *index*, for retrieving objects on the basis of p-relations. A+ *index* is a graph index where each global-key is represented by one node, and there are two types of edges connecting global-keys, representing *identity* and *matching* p-relations.

Example 6: Fig. 3 represents the A+ *index* of the polystore in Fig. 1. The double lines represent the identity p-relations, while the solid lines represent matching p-relations. On each edge, we keep the probability of the corresponding p-relation. For instance, the node representing the object whose global-key is catalogue.albums.d1 is connected to the node with global-key transactions.inventory.a32 with a probability of 0.9 (i.e. catalogue.albums.d1 $\sim_{0.9}$ transactions.inventory.a32).

C. Maintenance of the A+ index

a) Insertion of p-relations: To facilitate the execution of the augmentation operator at runtime, we enforce the CONSISTENCY CONDITION in the A+ *index*, and therefore we apply the transitivity property when an identity p-relation is inserted.

Example 7: In Fig. 4(a) a new identity is added to the A+ *index*, between global-key catalogue.albums.d1 and global-key discount.drop.k1:cure:wish. For the transitivity property, there is also an identity between catalogue.albums.d1 and all the identities of discount.drop.k1:cure:wish, that in this case, is with transactions.inventory.a32. We materialize this inferred p-relation in the graph as in Fig. 4(b) by considering the product 0.8×0.85 of the two connecting edges as probability, i.e. 0.68.



(a) New identity added

(b) Addition of identity p-relations for transitivity

Fig. 4. Maintenance of identity p-relations.

b) Deletion of p-relations: We adopt a lazy approach for deleting p-relations, that is, an object o is removed from the A+ *index* only if it turns out that, during the application of the augmentation construct, o is no longer present in the polystore. This is done by deleting the corresponding node in the A+ *index* and all its incident edges. We opted for a strategy that, when a p-relation x is deleted, the p-relations inferred via x are kept in the A+ *index*. In order to cover those use cases that require data oblivion, we will embed a lineage system that allows cascading deletions of inferred p-relations.

D. Collector

The collector is the component that collects p-relations whenever an A+ *index* is not already provided. In principle, any known technique for record linkage or, more generally, for identifying data relationships across multiple data sources can be used in the collector. The A+ *index* used in Section VII is generated by using state-of-the-art approaches to record linkage as black boxes. Record linkage techniques operate typically in two main phases: *blocking* and *pairwise matching*. We used BLAST [28] for non-supervised blocking of data objects, that is, to partition the data objects of the polystore in blocks so that the subsequent phase is restricted to the objects of the same block, and thus it is more efficient. BLAST fits our needs since it does not require any pre-existing knowledge of data sources. Pairwise matching is then performed using DuKe[2]. This tool has many built-in comparators and a genetic algorithm that we have used for tuning the configuration. On the basis of the final matching score provided by DuKe, we decide whether a pair of data objects forms a p-relation, and if yes, if it is an identity or a matching p-relation. In addition, we consider the rule under which, two different data object belonging to the same dataset cannot participate to an identity p-relation with the same object in a different database. This is because we assume that deduplication remains a local responsibility. If this is not the case after the *pairwise matching* phase, we keep the p-relations with higher probability only.

[2] https://github.com/larsga/Duke

a) Promotion of p-relations: In addition to the method mentioned above that can create an A⁺*index* from scratch, we have defined a simple, yet effective, learning mechanism that adds matching p-relations depending on the behavior of users when they access a polystore in exploratory mode.

We keep track in an internal repository, called $\mathcal{D}_P$, the *full paths* of the A⁺*index* that are traversed by users during augmented exploration. By full path we mean a sequence $t = v_0, v_1, \ldots, v_k$ of nodes of the A⁺*index* (with $k > 1$) such that v_0 contains the initial object of a user exploration and v_k the final object. $\mathcal{D}_P$ also stores the number of times that each full path t is traversed. The idea is that when the number of visits of a path t reaches a threshold τ_t, an edge (i.e. a matching p-relation x) between v_0 and v_k, if not yet present, is added in the A⁺*index*. The probability of x is computed by the average of all probabilities in the edges of t.

The rationale of this method is that a path visited many times highlights an interesting way to explore the polystore and so we try to anticipate user's intention by creating a shortcut. The threshold τ_t is chosen in such a way that its value decreases as the length of the path increases since the longer is a path the less and likely it is to be traversed.

Example 8: Figure 5 shows a full path in the A⁺*index* whose number of visits exceeds the threshold for a path of length three, and so a new edge is added between the ends of the path. It follows that we have inferred a matching relation between the data object in $v3$ and the data object in $v5$.

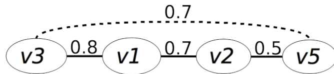


Fig. 5. Promotion of p-relations.

IV. AUGMENTERS

A. Network-efficient augmenter

Polystores are often deployed in a distributed environment, where network traffic has a significant impact on the overall performance of query answering. Augmentation, in particular, generates a non-negligible traffic by executing many local queries over the polystore, each one requesting a single data object. We implemented a BATCH augmenter that groups global keys by target database and submits them in one query. Next, BATCH arranges returned data objects to produce the answer. This batching mechanism tends to minimize the number of queries over the polystore, and so it also limits the burden of communication roundtrip on the overall execution. BATCH uses the parameter BATCH_SIZE that holds the maximum number of global keys per query.

In Fig. 6(b) we show the process of the BATCH augmenter in a graphical fashion on the same augmented query answering represented in Fig. 6(a). Global keys are grouped by store, as represented by the dotted internal boxes, and are retrieved with one query once the corresponding group reaches the BATCH_SIZE limit or when the process terminates. In the example, we set BATCH_SIZE = 4 and only one query per database is submitted, resulting in six queries less than the sequential augmentation (i.e. 5 instead of 11).

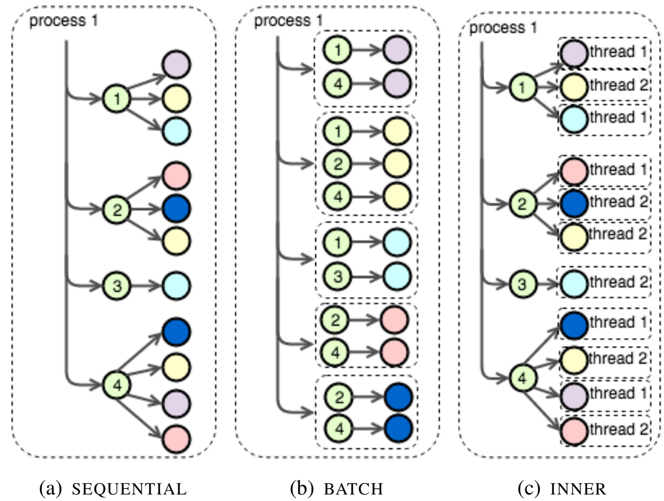


Fig. 6. Augmenters.

B. CPU-efficient augmenter

Augmented answers include data objects coming from different databases and so local queries can be submitted in parallel. We have designed a few strategies that leverage the multi-core nature of modern CPUs by assigning independent queries to parallel threads. These strategies are implemented in different augmenters, all parameterized with THREADS_SIZE, the maximum number of simultaneous running threads.

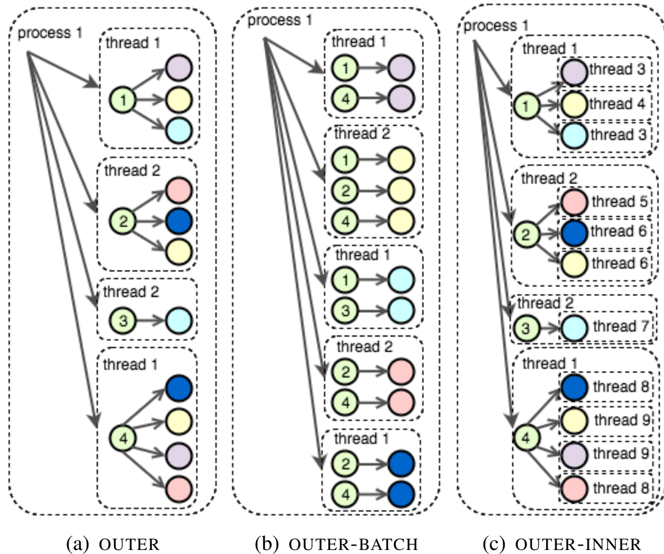


Fig. 7. Outer concurrency based augmenters.

a) Inner concurrency: This strategy exploits the observation that objects sharing an identity relation can be retrieved in parallel. In Fig. 6(c) we show this augmentation with THREADS_SIZE = 2 on the example in Fig. 6(a). The main process iterates over the result of the local query and, for each

object in the result, two threads compute the augmentation. This augmenter is very efficient for augmented exploration, in which a single result at a time needs to be augmented.

b) Outer concurrency: Differently from the previous strategy, the OUTER augmenter parallelizes the computation over the result of the local query. As shown in Fig. 7(a), the main process of OUTER iterates over the results in the result launching a thread for each of them without waiting for their completion. Then, each thread retrieves all objects related to the result in a sequential way.

c) Outer-Batch concurrency: The OUTER-BATCH augmenter combines multi-threading with batching. Differently from BATCH, the groups of global keys are processed by several threads. The main advantage here is that the main process can continue filling these groups while threads are taking care of query execution. This augmenter is parameterized with both THREADS_SIZE and BATCH_SIZE. In Fig. 7(b) we show the augmented process of the OUTER-BATCH with BATCH_SIZE = 2 and THREADS_SIZE = 4.

d) Outer-Inner concurrency: The OUTER-INNER augmenter tries to benefit from both "inner" and "outer" concurrency. The number of available threads, i.e. THREADS_SIZE, are used for the two levels of parallelism. It follows that $\frac{\text{THREADS_SIZE}}{2}$ threads process the results of the original answer in parallel, and further $\frac{\text{THREADS_SIZE}}{2}$ threads perform the augmentation for each result. Of course, this strategy tends to create many threads because of many simultaneous inner parallelizations. In Fig. 7(c) we show the augmentation process in OUTER-INNER with THREADS_SIZE = 4.

C. Memory-efficient strategies

All augmenters rely on a caching mechanism with a LRU policy that allows the fast access to the last accessed data objects by means of their global-key. The cache is implemented using Ehcache[3] with a suitable choice of CACHE_SIZE, the maximum number of objects in the cache. At runtime, we check whether the data object is already in the cache before asking for it to the polystore. Caching is potentially useful in two cases: (i) with augmented exploration, where the user accesses objects that were likely retrieved in previous queries, and (ii) with queries having level > 0, where augmented results of the same answer can overlap.

V. Adaptive Augmentation

QUEPA can run with different configurations. A configuration is a combination of the augmenter in use, CACHE_SIZE and, if needed, BATCH_SIZE and THREADS_SIZE. As the experiments in Section VII point out, none of the various configurations of QUEPA outperform the others in all possible scenarios. For example, some configuration excels on huge queries only, while others excel in a distributed environment. It follows that an optimizer is needed to choose the right augmenter and its parameterization in any possible situation.

Traditional cost-based optimizers are difficult to implement in a polystore because we might not have enough knowledge about each database system in play. Therefore, we designed an ADAPTIVE, rule-based optimizer to dynamically predict the best configuration according to the query and the polystore characteristics. It relies on a machine learning technique that generates rules able to select a well-performing configuration for the augmentation. The full process is as follows.

Phase 1 - Logs collection. We keep the logs of the completed augmentation runs. They include QUEPA parameters such as BATCH_SIZE or THREADS_SIZE, the overall execution time and the characteristics of the query (i.e. target database, number of original data objects in the result, number of augmented data objects). All these historical logs form our *training set*. In general, the larger is the training set, the higher is the accuracy of the trained models. When the training set is too small, we run, in background, previously executed queries with different configurations or we execute random queries against the polystore.

Phase 2 - Training. We train the following models:

- T_1 : a decision tree to decide the augmenter to use among those available (e.g., OUTER, INNER, BATCH, etc.). The tree is trained with the C4.5 algorithm [27];
- T_2 : a regression tree to decide BATCH_SIZE whenever T_1 selects OUTER-BATCH or BATCH. As we use Weka[4], this tree is trained with the REPTree algorithm [26];
- T_3 : a regression tree to decide THREADS_SIZE whenever a concurrent augmenter is selected by T_1. This is also trained with the REPTree algorithm;
- T_4 : a regression tree to decide CACHE_SIZE. This is trained with the REPTree algorithm.

The training of the models can be done periodically when a fixed number of run logs are added to the training set.

Phase 3 - Prediction. Given a query, we use our models to predict the parameters of QUEPA on how to augment the query. First, we determine with T_1 which augmenter we have to use. Then, according to the result, we use T_2 and T_3 for BATCH_SIZE and THREADS_SIZE. Finally, T_4 is used to decide the CACHE_SIZE. Since the benefits of the cache are spread over all future queries to run and not only on the next one, it has not much sense to change continuously the CACHE_SIZE. For example, increasing a lot CACHE_SIZE would just insert many empty cache slots. Rather, we want to determine slight variations of CACHE_SIZE that adapt to the queries currently being issued by the user.

The variation is calculated in the following way. We consider the CURRENT_CACHE_SIZE and the PREDICTED_CACHE_SIZE determined by T_4. Then, we use the formula $\frac{(\text{PREDICTED_CACHE_SIZE} - \text{CURRENT_CACHE_SIZE})}{10}$, where 10 is an arbitrary value set by us experimentally.

Fig. 8 shows an example of the decision tree T_1. When a new query has to be executed, we navigate the tree from the root to a leaf according to the characteristics of our setting.

[3] http://www.ehcache.org/

[4] http://www.cs.waikato.ac.nz/ml/weka/

The leaves indicate the final decision, i.e. the augmenters to choose.

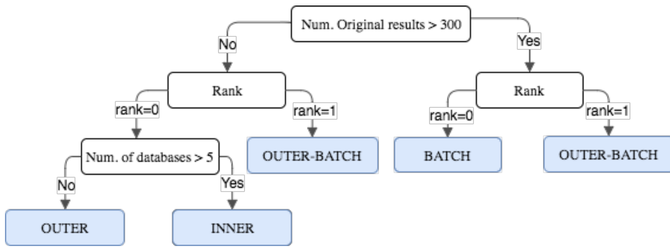


Fig. 8. An example of decision tree T_1.

VI. Related Work

A. Data Integration in Polystores

The integration of heterogeneous databases is typically achieved with a middleware layer that manages a global schema that is the reference for queries. Existing middleware solutions are basically different recipes of the same dish, each one flavouring with the basic ingredients: data modeling, query optimization, storage organization and query languages. For simplicity, we separate the discussion into different categories.

a) Tightly-coupled integration with a multi-model architecture: Multi-model architectures address the heterogeneity issues in a "one size fits all" fashion. They basically propose an all-in-one platform to avoid the proliferation of engines in use within an organization. They unify, in a tightly-coupled architecture, the storage and the query optimizer for all supported data models [2], [4], [21]. AsterixDB relies on a flexible data model with open and closed types to subsume both structured and semi-structured data [2]. ArangoDB [4] and OrientDB [21] have similar architectures supporting both graph and document-based data. ArangoDB includes also key-value structured data. Unified architectures have the advantage to natively use a single query language, but on the other hand, they usually perform worse than a specialized system.

b) Loosely-coupled integration with common interfaces: Most of the approaches provide a common interface to heterogenous database systems [3], [6], [17], [20], [33], [34]. They readapt strategies used for data mediators and database federation [15], [24]. Apache Metamodel [3], SQL++ [20] and CloudMdsQL [17] provide an interface to several types of database systems and are equipped with a SQL-like query language. The user specifies the global schema along with the definition of views over each data source. The middleware optimizes query execution taking into account the specificity of each engine for "pushing down" query fragments to the stores. Query optimization and query rewriting in modern polystores are still open areas of research, where the main challenge is to generalize the middleware for capturing database engines [23]. RACO is a middleware for heterogeneous query languages that is used in Myria for federating multiple backend systems [34]. SOS [6] proposes a common interface to NoSQL engines that is based on basic access primitives, though it does not deal with data integration. UnQL proposes an experanto query language for querying different database systems [33]. Having a common interface is good for switching from one system to another, but users have to sacrifice performance of the local system. In addition, sharing all the features would require a major redesign of each database engine.

c) Other loosely-coupled integration architecture: Polyglot architectures are also used in analytical scenarios [1], [12], [13]. BigDAWG federates heterogeneous engines with a "scope & cast" strategy that makes it possible to run analytics queries over large-scale polystores [12]. It considers islands of information defined by the user where data can be migrated from a system to another. The system decides in which engine query execution has to take place, possibly taking into account the cost of data migration. RHEEM optimizes and processes big data workloads when multiple processing engines are available. It offers a set of primitives that are split into more granular operations, each one executed on the system that is more suitable for each primitive over each workload [1]. MuSQLE is a SQL multi-engine optimizer based on Spark that interfaces with the polystore via APIs [13].

Our approach to data access in polystores differs from all the above and applies well to contexts where we do not have a global schema of the polystore. With respect to a short paper illustrating a demo of the first implementation of QUEPA [18], we present here the precise semantics of the augmentation-based operators and all the technical details under the hood that, according to our experimental campaign, make the approach viable, efficient, and replicable. Moreover, the techniques proposed in this paper for efficiently using memory, CPU and network can provide a contribution to the general problem of supporting integration and query answering in polystores.

B. Discovery and Exploratory Search

Exploratory search emerged a decade ago for empowering search interfaces with browsing features. Nowadays, the ability to discover interesting information is of major relevance in data science where users are not fully aware of the massive quantities of data available. Many approaches leverage on dependencies and correlations between attributes to suggest portions of the database to explore [7], [9], [29], [35]. DBExplorer proposes a summarization technique that considers conditional dependencies between attribute values, both within and across attributes [29]. Buoncristiano et al. consider views of the database organized in a lattice [7]. Das et al. extend a search engine with a faceted search technique to return the most interesting facets (i.e. surprising and unexpected) with respect to a set of documents matching input keywords [9]. InfoGather provides entity augmentation to collect relevant information from Web tables [35].

All these approaches help in exploring portions of the same dataset but, differently from ours, are difficult to apply over polystores.

C. Linked data and query relaxation

Our approach is also related to the problem of querying linked data [16] since in our approach a polystore can be

viewed as an RDF-like graph in which the nodes are data objects, possibly stored in different databases, and the links represent the p-relations between them. However, our scenario is rather different since in polystores data are not freely accessible, as it happens on the Web, but are stored in internal databases for which a global view is usually not available (and so SPARQL-like queries are not possible). Moreover, and more important, in our approach the graph is transparent to the user as she can only perform augmented searches, using the local language, or augmented explorations, by clicking on retrieved data. In addition, it should be observed that the augmentation construct provide a form of query relaxation [10], [19], [25], in that it allows the user to extend the exact answer to a query with further interesting data.

VII. Empirical Evaluation

A. Setting

a) Polystore setting: We used a polystore compliant to the running example in Fig. 1, in which the warehouse department has a MongoDB v.3.0.12 instance, the sales department uses MySQL v.5.6.30, the marketing department relies on Neo4j v.3.1.0 and the shared key-value store is on Redis v.3.0.5. However, we point out that the findings of this experimentation are invariant with respect to the choice of the systems composing the polystore.

We used real data to populate the databases. Songs and their similarities are taken from Last.fm dataset[5], albums are reconstructed from the songs by using the MusicBrainz web service [31]. Customers' profiles, sales details, and discounts are generated synthetically. The size of the polystore is as follows: around 8 million json documents in MongoDB, around 20 million tuples in MySQL, around 4 million entries in Redis, around 5 million nodes and 20 millions of edges in the similar-items of Neo4j.

With the techniques and tools described in Section III-D we populated the A$^+$*index* with almost 100 millions of nodes and 500 millions of edges. We set the thresholds experimentally as follows. Matching p-relations are those having probability included between 0.6 and 0.89, while identity p-relations have probability equal or higher than 0.9. However, the quality and the semantics of the generated p-relations are irrelevant to the purpose of this experimentation, which is focused on performance and scalability of the augmentation construct.

We replicated the databases, and updated the A$^+$*index* accordingly, to create different but comparable versions of polystores with increasing size. Each database was replicated three times with the exception of Redis that remains a single instance in all the polystore versions. Each replica runs as a separate instance and so, from QUEPA's perspective, each replica is seen as a completely different database.

The polystores are deployed in a centralized and in a distributed environment using Amazon EC2 machines. The centralized environment is on a *m4.4xlarge* machine equipped with 16 vCPU, 64GB and running a 2,3 GHz Intel Xeon with 18 cores. The distributed environment runs QUEPA on a *m4.4xlarge* machine and the replicated databases on different *t2.medium* machines. Each machine is placed in a different region to increase network latency.

b) Test bed setting: We created a set of queries in the following way. For each of the four databases, we consider queries with different result size: they retrieve 100, 500, 1.000, 5.000 and 10.000 objects. The number of data objects within the augmented answers of these queries over the polystores described above range from 400 to 1 million. When experiments are shown with respect to the query size, we show the average execution time of the corresponding queries on each target database.

The graph of the A$^+$*index* is uniformly dense and so it does not bias the analysis of performance with respect to the query size. In other words, queries of the same size return answers with a comparable number of data objects and the number of data objects increases linearly with the number of results in the original query. All queries are submitted with augmented search since augmented exploration is just a simplified procedure that would have been unfair to compare against competitors.

We conducted cold and warm cache experiments. The cold-cache runs were performed dropping both the operating system cache (by executing `/bin/sync` and `echo 3 > /proc/sys/vm/drop_caches`) and the DBMSs caches (by restarting them before executing the query). The running time of each warm-cache run was taken from a subsequent execution of the corresponding cold-cache run. All the times reported are "end-to-end" query performance and include the outputting of results. Every test was executed three times, out of which we consider the average.

c) Middleware tools setting: We compare QUEPA against publicly available middleware tools. We used Apache Metamodel 4.5.4[6], Talend Open Studio for Big Data 6.2[7] and ArangoDB 3.1.7[8]. Apache Metamodel is a representative of the loosely-coupled integration interfaces and allowed us to integrate MySQL, Neo4j and MongoDB instances in our polystore (that is, Redis is not supported). On Metamodel, we implemented the augmentation process in two different ways. The first uses native operators based on joins, while the second simulates the augmentation algorithm of QUEPA. Talend is a representative of the traditional data integration techniques. We created a workflow for executing augmentation over the polystore by using Neo4J, MySQL and MongoDB connectors (that is, Redis is not supported). The workflow was compiled and used as a standalone tool. ArangoDB is an in-memory database management system that represents multi-model architectures. It allowed us to import our key-value, graph and document databases (that is, relational databases are not supported). We stored the A$^+$*index* and the polystore in ArangoDB and, as for MetaModel, we implemented the

[5]http://labrosa.ee.columbia.edu/millionsong/lastfm

[6]http://metamodel.apache.org/

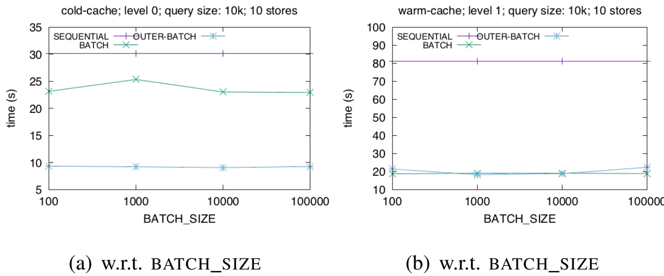
[7]https://www.talend.com/download/talend-open-studio

[8]https://www.arangodb.com/

augmentation on ArangoDB in a native way and in QUEPA-style. The former implementation executes the augmentation with a single AQL (ArangoDB Query Language) query. The latter implementation uses a modified version of QUEPA augmentation algorithm that interacts with ArangoDB just to access data objects and the A+ *index*.

B. Performance of QUEPA

In this section, we evaluate the configurations and the optimization techniques implemented in QUEPA and discussed in Section III and in Section IV. For a comprehensive view, we opted to show both cold-cache and warm-cache queries, at level 0 and level 1, respectively.



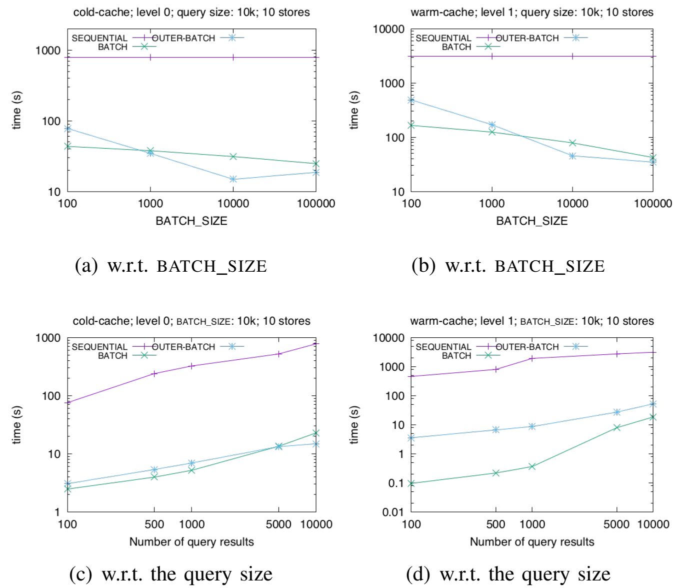
(a) w.r.t. BATCH_SIZE (b) w.r.t. BATCH_SIZE

Fig. 9. Scalability of augmentation with batching.

a) Network-based optimizations: We show the effect of batching when varying BATCH_SIZE for both BATCH and OUTER-BATCH over original queries with 10.000 results in a polystore with 10 stores (see Fig. 9(a) and Fig. 9(b) taking into consideration the log scale of the x-axis). BATCH is more sensitive to BATCH_SIZE than OUTER-BATCH, which on the other hand can also benefit from multi-threading (for this campaign THREADS_SIZE = 4). However, the multi-threading effect tends to vanish during warm-cache run.

We tested the batching in the distributed deployment as well, where the network latency reaches, in some cases, few hundred milliseconds. Considering the log scale, we observe the strong boost of the batching compared to the sequential counterpart (see Fig. 10(a) and Fig. 10(b)). It is also evident how batching is more effective in distributed rather than in centralized polystores. The improvement increases as BATCH_SIZE also increases. For high values of BATCH_SIZE, BATCH and OUTER-BATCH tend to behave similarly, that is, the effect of batching can, to a certain extent, dissolve the benefit of multi-threading. The batching has not only the advantage of decreasing execution time, but it scales better with larger inputs than other systems (see Fig. 10(c) and Fig. 10(d)).

b) CPU-based optimizations: We conducted a campaign of experiments to test the impact of multi-threading. Initially, as shown in Fig. 11(a) and Fig. 11(b), we check the behaviour of augmenters when varying THREADS_SIZE. All concurrent augmenters decrease their performance when THREADS_SIZE increases. Generally, INNER performs worse than the outer concurrency-based augmenters because it is limited by the number of different stores, which is lower than the number of results in the original query. We can see that all



(a) w.r.t. BATCH_SIZE (b) w.r.t. BATCH_SIZE

(c) w.r.t. the query size (d) w.r.t. the query size

Fig. 10. Distributed augmentation with batching.

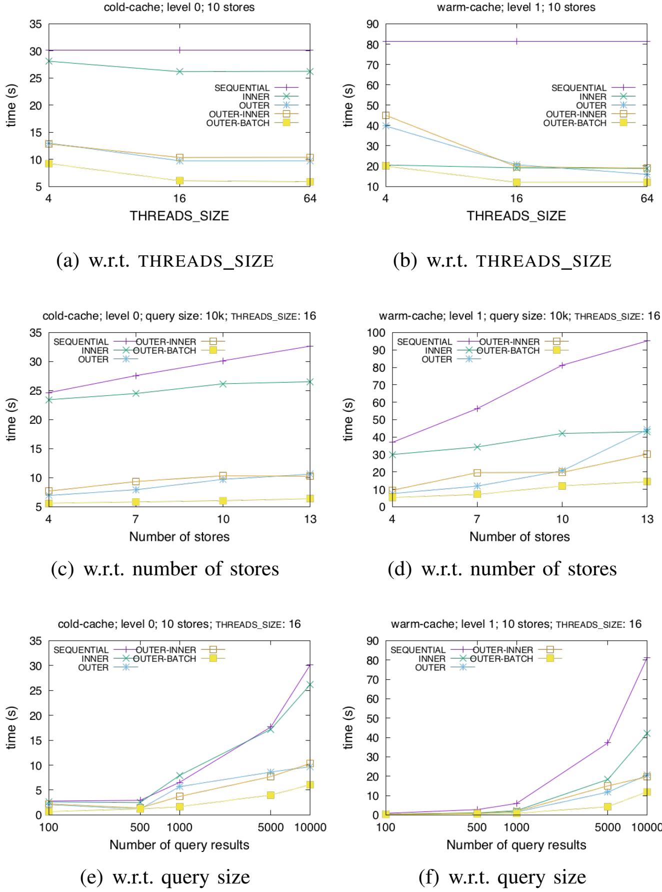
augmenters speed-up until 16 running threads and stabilize afterward. From further experiments not present in this paper and conducted on less performing machines, we witness that this behavior is due to the cores of the underlying machine (i.e. 18 in our case). Therefore, there is further room for improving performance on infrastructures with more cores. We also notice that, sometimes, the bottleneck is represented by the underlying stores that cannot manage many concurrent requests. This was due to the fact that the stores are running on a machine slower than the machine where QUEPA is running.

Figures 11(c)-11(d)-11(e)-11(f) enrich our experimentation by showing the scalability of the augmenters when the size of the queries and the number of underlying databases increases. There is only one run in which the concurrent augmenters perform worse than the sequential. However, in best case scenarios, where the query size is much smaller and the number of stores is reduced, SEQUENTIAL performs better than others because of the overhead of creating and synchronizing threads. Overall, OUTER-BATCH is the best and INNER is the worst.

c) Memory-based optimizations: Another campaign of tests was conducted to evaluate the effect of caching. We do not illustrate these tests here for lack of space but we report that augmentation is, in the centralized deployment, less sensitive to CACHE_SIZE than to other parameters. This is basically due to the fact that each local database system has its own caching mechanism, making QUEPA's cache partly redundant. Caching resulted beneficial in the distributed deployment where machines are not co-located since, as well as batching, it allows to "save" inter-machine communication.

C. Quality of the Optimization

In this section, we show the accuracy of ADAPTIVE, which we have trained with the logs of almost 2 million runs. We compare ADAPTIVE against a HUMAN optimizer and a



(a) w.r.t. THREADS_SIZE (b) w.r.t. THREADS_SIZE

(c) w.r.t. number of stores (d) w.r.t. number of stores

(e) w.r.t. query size (f) w.r.t. query size

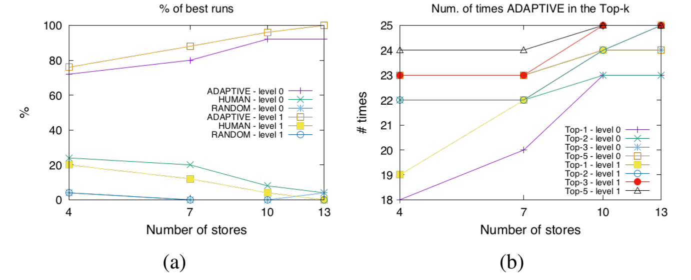
Fig. 11. Scalability of concurrent augmenters.

RANDOM optimizer. The campaign was planned as follows. We have generated 25 queries of a different kind that were not present in the training set. Each query is run on each of our four polystore variants (4, 7, 10 and 13 databases) and for both level 0 and level 1 augmentation search. This means that for each query we have 8 runs.

For the HUMAN optimizer, we defined the configuration for each run that could, in our opinion, result to be the most performing. A configuration consists of THREAD_SIZE, BATCH_SIZE and CACHE_SIZE. Each configuration is executed for each of the six available augmenters. In addition, we defined a random configuration for each run in order to emulate a RANDOM optimizer. Finally, we have another run whose configuration is determined by ADAPTIVE. Note that the use of CACHE_SIZE in this campaign of experiments work in the same way it is described in Section V. For this reason, we first run all the HUMAN runs, followed by RANDOM and then ADAPTIVE.

For each configuration, we need to select the best performing run out of the 13 (i.e. 1 for ADAPTIVE and 6 for both HUMAN and RANDOM).

In Fig. 12(a) we compare the number of times that an optimizer is the best. Although the number of candidates for ADAPTIVE was six times lower than the other optimizers, it was the best in most of the cases. In Fig. 12(b) we show the number of times that the ADAPTIVE run was in the top-1, top-2, top-3 and top-5 runs. ADAPTIVE is always able to find a good configuration for the query. The accuracy of ADAPTIVE increases as the number of databases increases because the differences of execution times between configurations increase, thus making it easier for the decision trees to split the domain of the parameters.

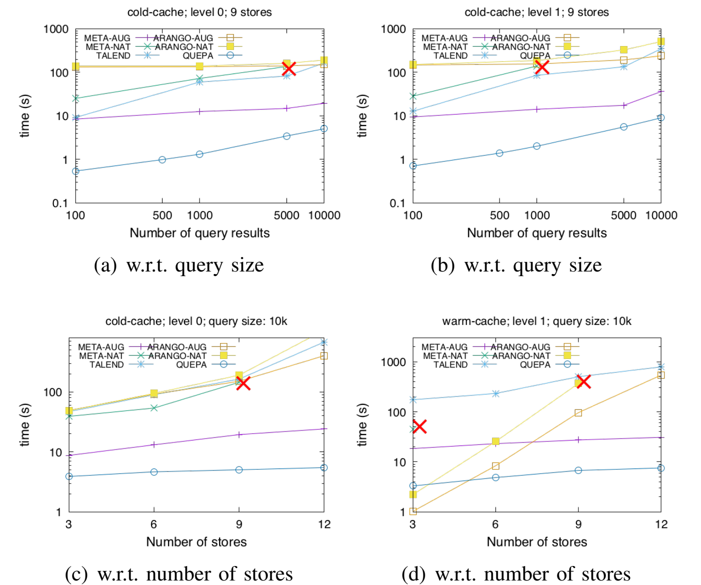


(a) (b)

Fig. 12. Accuracy of the ADAPTIVE augmenter optimization.

D. Comparison with middleware approaches

In this section, we show the performance of QUEPA compared to middleware layer approaches running in their default configuration. For QUEPA, we use the default augmenter, ADAPTIVE, which is able to select a convenient configuration for running the query. The suffix "AUG" applied to Metamodel (i.e. META) and to ArangoDB (i.e. ARANGO) stands for the implementation of augmentation that simulates our algorithm, while the suffix "NAT" stands for an implementation that uses native operators. We indicate the points after which the runs go out-of-memory with a red 'X'.



(a) w.r.t. query size (b) w.r.t. query size

(c) w.r.t. number of stores (d) w.r.t. number of stores

Fig. 13. Comparison vs middleware approaches.

In Fig. 13(a) and Fig. 13(b) we show the scalability of all

systems with respect to the size of the query (note the log scale for both axes) in a polystore with 9 stores. We do not show the performance on top of the polystore with 12 databases because META-NAT and ARANGO-NAT go out-of-memory with most of the queries. As we can see, QUEPA is the most performing. The two versions of ArangoDB are the least performing because they need to warm up at start-up, with ARANGO-AUG performing slightly better than ARANGO-NAT. Metamodel scales only with the augmentation operator and goes often out-of-memory otherwise. Talend behaves similarly to META-AUG for small queries but overall the trend presents the steepest slope.

It is interesting to analyse the scalability of the approaches over the number of databases involved in the polystore (see Fig. 13(c) and Fig. 13(d)). Given that ArangoDB is an in-memory database, it performs well on warm-cache runs, but its performance decrease significantly when we add databases to the polystore and, indeed, it falls often into out-of-memory situations. Metamodel is able to scale similarly to QUEPA when emulating the augmentation operator and it is not practicable with native operators. Talend confirms previous experiments showing, again, a steep trend. QUEPA scales smoothly as the polystore becomes larger, thus demonstrating a nice way to leverage parallelism.

The takeaway from this experimentation is that augmentation is an efficient way to access a polystore that can be implemented on existing polystore engines. In addition, the augmentation operator allows for a series of tailored optimizations that can be used to further enhance performance.

VIII. Conclusion and Future Work

In this paper, we have proposed a new approach for querying and exploring a polystore that relies on query augmentation, a novel data manipulation operator for polystores that allows the automatic expansion of the answer to a query on a local datastore with related data stored in the rest of the polystore.

The computation of this operator does not require middleware layers, global schemas or universal data models. The augmentation operator can be used for implementing augmented search and augmented exploration, two practical methods for data access that support information discovery and data integration in polystores. We have also illustrated the design and the optimization techniques of a system that fully implements our approach in an efficient way. A number of experiments have confirmed that the approach is feasible and scales smoothly over very large polystores.

As a direction of future work, we would like to extend augmentation to data analytics scenarios. We are also studying more performing strategies to implement our A^{+} *index*.

References

[1] D. Agrawal et al. Rheem: Enabling multi-platform task execution. In *SIGMOD*, pages 2069–2072, 2016.
[2] S. Alsubaiee et al. AsterixDB: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.
[3] Apache MetaModel. http://metamodel.apache.org/, (accessed September, 2017).
[4] ArangoDB. https://www.arangodb.com/, (accessed September, 2017).
[5] P. Atzeni, F. Bugiotti, L. Cabibbo, and R. Torlone. Data modeling in the nosql world. *Computer Standards and Interfaces*, 2016.
[6] P. Atzeni, F. Bugiotti, and L. Rossi. Uniform access to NoSQL systems. *Inf. Syst.*, 43:117–133, 2014.
[7] M. Buoncristiano, G. Mecca, E. Quintarelli, M. Roveri, D. Santoro, and L. Tanca. Database challenges for exploratory computing. *SIGMOD Record*, 44(2):17–22, 2015.
[8] U. Çetintemel, M. Cherniack, J. DeBrabant, Y. Diao, K. Dimitriadou, A. Kalinin, O. Papaemmanouil, and S. B. Zdonik. Query steering for interactive data exploration. In *CIDR*, 2013.
[9] D. Dash, J. Rao, N. Megiddo, A. Ailamaki, and G. M. Lohman. Dynamic faceted search for discovery-driven analysis. In *CIKM*, pages 3–12, 2008.
[10] R. De Virgilio, A. Maccioni, and R. Torlone. Approximate querying of RDF graphs via path alignment. *Distributed and Parallel Databases*, 33(4):555–581, 2015.
[11] K. Dimitriadou, O. Papaemmanouil, and Y. Diao. Explore-by-example: an automatic query steering framework for interactive data exploration. In *SIGMOD*, pages 517–528, 2014.
[12] J. Duggan et al. The BigDAWG polystore system. *SIGMOD Record*, 44(2):11–16, 2015.
[13] V. Giannakouris, N. Papailiou, D. Tsoumakos, and N. Koziris. MuSQLE: Distributed SQL query execution over multiple engine environments. In *BigData 2016*, pages 452–461, 2016.
[14] L. M. Haas. The power behind the throne: Information integration in the age of data-driven discovery. In *SIGMOD*, page 661, 2015.
[15] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *VLDB*, pages 276–285, 1997.
[16] O. Hartig. An overview on execution strategies for linked data queries. *Datenbank-Spektrum*, 13(2):89–99, 2013.
[17] B. Kolev, C. Bondiombouy, P. Valduriez, R. Jiménez-Peris, R. Pau, and J. Pereira. The CloudMdsQL Multistore System. In *SIGMOD*, pages 2113–2116, 2016.
[18] A. Maccioni, E. Basili, and R. Torlone. QUEPA: QUerying and exploring a polystore by augmentation. In *SIGMOD*, pages 2133–2136, 2016.
[19] D. Martinenghi and R. Torlone. Taxonomy-based relaxation of query answering in relational databases. *VLDB J.*, 23(5):747–769, 2014.
[20] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases. *CoRR*, abs/1405.3631, 2014.
[21] OrientDB. http://orientdb.com/, (accessed September, 2017).
[22] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64:1183–1210, 12 1969.
[23] Y. Papakonstantinou. Polystore query rewriting: The challenges of variety. In *EDBT/ICDT Workshops*, 2016.
[24] Y. Papakonstantinou, A. Gupta, and L. M. Haas. Capabilities-based query rewriting in mediator systems. *Distributed and Parallel Databases*, 6(1):73–110, 1998.
[25] A. Poulovassilis, P. Selmer, and P. T. Wood. Approximation and relaxation of semantic web path queries. *J. Web Sem.*, 40:1–21, 2016.
[26] J. R. Quinlan. Simplifying decision trees. *International Journal of Man-Machine Studies*, 27(3):221–234, 1987.
[27] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
[28] G. Simonini, S. Bergamaschi, and H. V. Jagadish. BLAST: a loosely schema-aware meta-blocking approach for entity resolution. *PVLDB*, 9(12):1173–1184, 2016.
[29] M. Singh, M. J. Cafarella, and H. V. Jagadish. DBExplorer: Exploratory search in databases. In *EDBT*, pages 89–100, 2016.
[30] M. Stonebraker. The case for polystores. http://wp.sigmod.org/?p=1629, July, 2015.
[31] A. Swartz. Musicbrainz: A semantic web service. *IEEE Intelligent Systems*, 17(1):76–77, 2002.
[32] The DZone Guide To Data Persistence. https://dzone.com/guides/data-persistence-2, (accessed September, 2017).
[33] UnQL: Unstructured Data Query Language. http://www.couchbase.com/press-releases/unql-query-language, (accessed September, 2017).
[34] J. Wang et al. The myria big data management and analytics system and cloud services. In *CIDR*, 2017.
[35] M. Yakout, K. Ganjam, K. Chakrabarti, and S. Chaudhuri. InfoGather: entity augmentation and attribute discovery by holistic matching with web tables. In *SIGMOD*, pages 97–108, 2012.