Learning How to Optimize Data Access in Polystores

Antonio Maccioni[1(✉)] and Riccardo Torlone[2]

[1] Collective[i], New York City, USA
amaccioni@collectivei.com
[2] Roma Tre University, Rome, Italy
riccardo.torlone@uniroma3.it

Abstract. Polystores provide a loosely coupled integration of heterogeneous data sources based on the direct access, with the local language, to each storage engine for exploiting its distinctive features. In this framework, given the absence of a global schema, a common set of operators, and a unified data profile repository, it is hard to design efficient query optimizers. Recently, we have proposed QUEPA, a polystore system supporting *query augmentation*, a data access operator based on the automatic enrichment of the answer to a local query with related data in the rest of the polystore. This operator provides a lightweight mechanism for data integration and allows the use of the original query languages avoiding any query translation. However, since in a polystore we usually do not have access to the parameters used by query optimizers of the underlying datastores, the definition of an optimal query execution plan is a hard task, as traditional cost-based methods for query optimization cannot be used. For this reason, in the effort of building QUEPA, we have adopted a machine learning technique to optimize the way in which query augmentation is implemented at run-time. In this paper, after recalling the main features of QUEPA and of its architecture, we describe our approach to query optimization and highlight its effectiveness.

1 Introduction

The concept of polyglot persistence, which consists of using different database technologies to handle different data storage needs [15], is spreading within enterprises. Recent research has shown that, on average, each enterprise application relies on at least two or three different types of database engines [17].

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

© Springer Nature Switzerland AG 2019
V. Gadepally et al. (Eds.): DMAH 2019/Poly 2019, LNCS 11721, pp. 115–127, 2019.
https://doi.org/10.1007/978-3-030-33752-0_8

recommendations. In addition, there exists a key-value store containing discounts on products, which is shared among the three departments above.

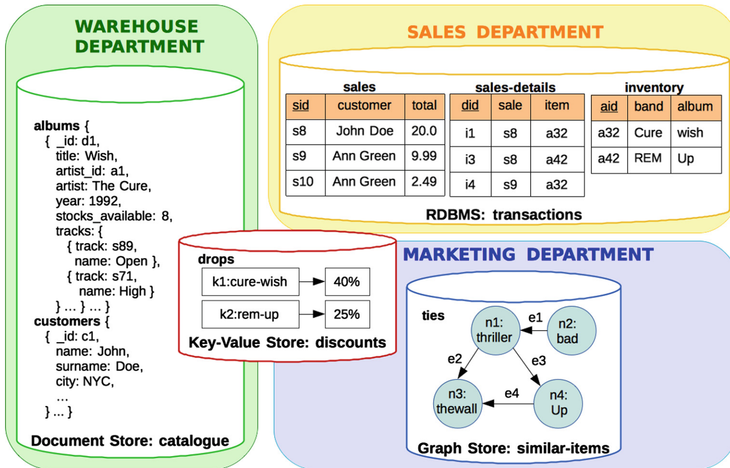


Fig. 1. A polyglot environment.

In this framework, it is of strategic importance to provide easy-to-use mechanisms for searching through all the available data [4]. 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 [2,11]. However, this solution adds computational overhead at runtime and, more importantly, hides the specific features that these systems were adopted for. In addition, it is hard to maintain, having an inherent complexity that increases significantly as new systems take part to the environment.

Polystore systems (or simply, polystores) have been proposed recently as an alternative solution for this scenario [16]. The basic idea is to provide a loosely coupled integration of data sources and allow the 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 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"* [16].

With the aim of providing a contribution to this problem, we have recently proposed (*data*) *augmentation* [9], a new construct for data manipulation in polystores that, given an object o taken from a database of a polystore, allows

the automatic retrieval of a set of objects that: (i) are stored elsewhere in the polystore and (ii) are somehow related with o, according to a simple notion of probabilistic relationship between objects in different datastores.

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 [1,3,5,11].

The augmentation construct was implemented in QUEPA [8], a system that provides an effective method to access the polystore called *augmented search*. 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 > $\Rightarrow$ (catalogue:{ title: Wish,
$\Downarrow$ artist_id: a1,
(discounts: 40%) artist: The Cure,
year: 1992,
... })

In an augmented 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.

As it happens in traditional query optimization, the best performances of query answering in QUEPA are achieved by properly tuning a series of parameters. Some of these parameters depends of the polystore setting and can be configured by the system administrator once, when she has enough knowledge on the underlying databases. Other parameters depends on the specific query workload (e.g., the selectivity of queries) that are more difficult to tune. In general, traditional cost-based optimizers are hard to implement in a polystore because we might not have enough knowledge about the parameters affecting the optimization of each database system in play.

We saw this limitation as the opportunity to experiment different optimization approaches. To this aim, we equipped the system with a rule-based optimizer to dynamically predict the best configuration according to the query and the polystore characteristics. It relies on machine learning algorithms that learn from previous query executions what is the best execution plan given an input query. The idea of using machine learning within query optimization was then also explored by Krishnan et al. [7] and Marcus et al. [10]. They adopt deep reinforcement learning for optimizing joins. Other relevant work also use machine learning to improve the indexing of data [6].

In our approach, we train a series of decision trees with the statistics gathered from previous queries. These trees are then used at query time to determine the values of the configuration parameters to be used by the query orchestrator. In this way, neither the user nor the sysadmin need to do any tuning manually. The experiments have confirmed the effectiveness of the approach.

In the rest of the paper, after a brief overview of our approach and of the system we have developed (Sect. 2), we illustrate the way in which we have implemented query augmentation, the main operator of QUEPA, and the adaptive technique we have devised for predicting the best query plan for a query involving augmentation (Sect. 3). We also illustrate some experimental results supporting the effectiveness of the optimization technique (Sect. 4) and sketch future directions of research (Sect. 5).

2 Augmented Access to Polystores

2.1 A Data Model for Polystores

We model a polystore as a set of databases stored in a variety of data management systems (relational, key-value, graph, etc.). A database consists of a set of *data collections* each of which is made of a set of *(data) objects*. An object is just a key-value pair. A tuple and a JSON document are examples of data objects in a relational database and in a document store, respectively. We assume that a data object in the polystore can be uniquely identified by means of a *global key* made of: its key, the data collection C it belongs to, and the database including C. Basically, this simple model captures any database system satisfying the minimum requirement that every stored data object can be identified and accessed by means of a key.

We also assume that data objects of possibly different databases of a polystore can be correlated by means of *p-relations* (for relations in a polystore). 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 $\sim$: 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.

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 [12], that is, algorithms able to score the likelihood that a pair of objects in different databases match.

2.2 Augmented Search

The augmentation construct 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.

Formally, the augmentation α^n of level $n \geq 0$ of a set of objects in a polystore $\mathcal{P}$ 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.

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. Formally, the *augmentation of level* $n \geq 0$ of a query Q over a database $\mathcal{D}$ of a polystore (expressed in the query language of the storage system used for $\mathcal{D}$), denoted by $Q(n)(\mathcal{D})$, consists in the augmentation of level $n \geq 0$ of the result of Q 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.

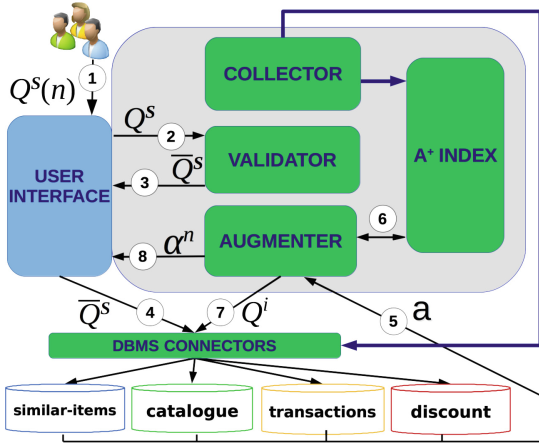


Fig. 2. Architecture of QUEPA.

2.3 Implementation

We have implemented our approach in a system called QUEPA. Its architecture is reported in Fig. 2 and includes the following main components:

- *Augmenter:* implements the augmentation operator and orchestrates augmented query answering.
- A^+ *index:* stores the p-relations between data objects in the polystore.
- *Collector:* this component is in charge of discovering, gathering and storing p-relations in the A^+ *index*.
- *Connectors:* they allow the communication with a specific database system by sending queries in the local language and returning the result.
- *Validator:* is used to assess whether a query can be augmented or not. 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 in augmented mode with level n (step ① in Fig. 2).

The *validator* first checks if the query is correct (step ②) and possibly rewrites it into $\overline{Q}$ (step ③) before its execution over the target database (step ④). The local answer **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 **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).

3 Efficient Implementation of Augmented Search

3.1 Augmenters

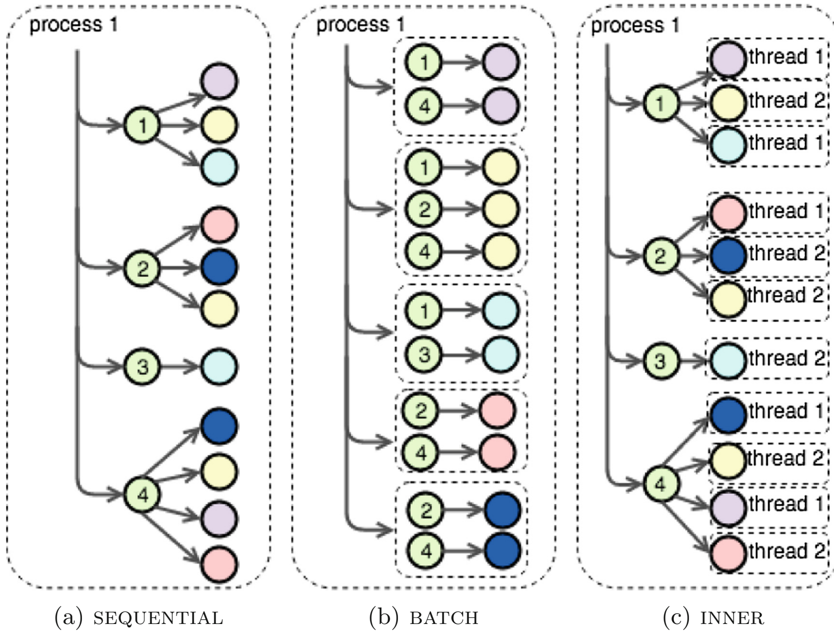
The augmentation operator is inherently distributed because it retrieves data from independent databases. We leverage that and other characteristics of this operator to make the augmentation more efficient.

Figure 3(a) illustrates the augmentation process done in a sequential fashion: 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.

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. 3(b) we show the process of the BATCH augmenter in a graphical fashion on the same augmented query answering represented in Fig. 3(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).

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.

Inner Concurrency. This strategy exploits the observation that objects sharing an identity relation can be retrieved in parallel. In Fig. 3(c) we show this augmentation with THREADS_SIZE = 2 on the example in Fig. 3(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.



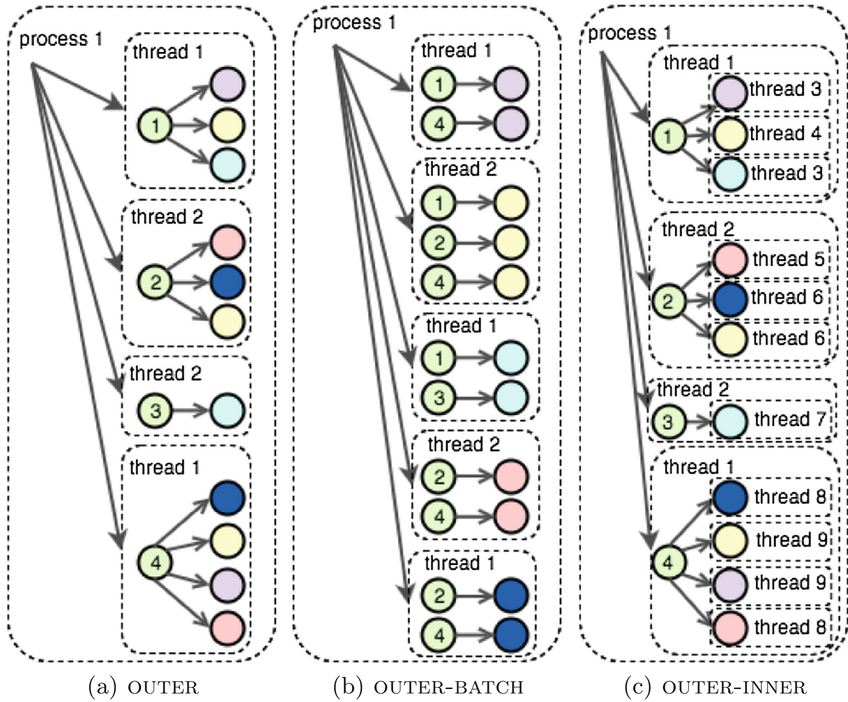
(a) SEQUENTIAL (b) BATCH (c) INNER

Fig. 3. Augmenters.

Outer Concurrency. Differently from the previous strategy, the OUTER augmenter parallelizes the computation over the result of the local query. As shown in Fig. 4(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.

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. 4(b) we show the augmented process of the OUTER-BATCH with BATCH_SIZE = 4 and THREADS_SIZE = 2.

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. 4(c) we show the augmentation process in OUTER-INNER with THREADS_SIZE = 4.



(a) OUTER (b) OUTER-BATCH (c) OUTER-INNER

Fig. 4. Outer concurrency based augmenters.

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[1] 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. The level represents the hops of distance in A^+ index between the objects of the original result set and the objects in the augmented result.

3.2 Adaptive Augmentation: Learning the Access Plans

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 Sect. 4 of [9] point out, none of the various configurations of QUEPA outperform the others in all possible scenarios.

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

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.

As we have observed in the Introduction, 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 [14];

  T_2: a regression tree to decide BATCH_SIZE whenever T_1 selects OUTER-BATCH or BATCH. As we use Weka[2], this tree is trained with the REPTree algorithm [13];

  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

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

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.

Figure 5 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. The leaves indicate the final decision, i.e. the augmenters to choose.

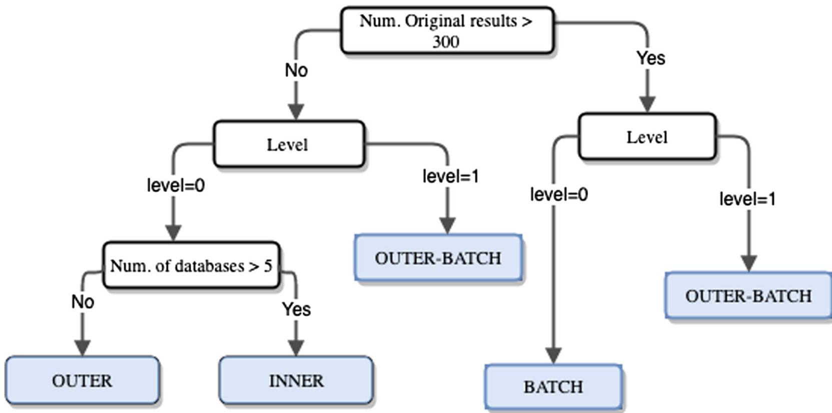


Fig. 5. An example of decision tree T_1.

4 Summary of Experiments

In this section we show the effectiveness of our learning mechanism only. The full report of the results is shown in [9].

The ADAPTIVE augmented has been trained with the logs of almost 2 million runs. We compare ADAPTIVE against a HUMAN optimizer and a 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 a polystore with a different number of databases (4, 7, 10 and 13).

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 Sect. 3.2. 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).

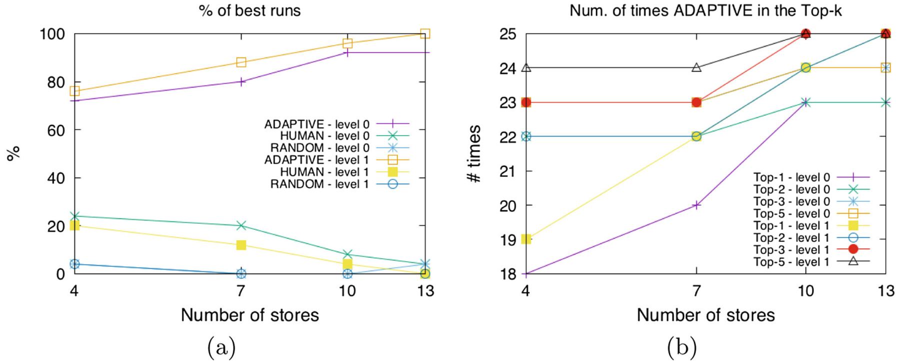


Fig. 6. Accuracy of the ADAPTIVE augmenter optimization.

In Fig. 6(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. 6(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.

5 Conclusion and Future Work

In this paper we have shown that machine learning can be used to optimize the access to data in a polystore. Indeed, as the database systems in a polystores are black boxes, a mechanisms that learns automatically the best way to exploit them with no knowledge of their internals can be very effective. In particular, we adopted this solution to optimize the query augmentation mechanism offered by our polystore system, QUEPA. Augmentation provides an effective tool for information discovery in heterogenous environments that, according to the polystore philosophy, does not require any query translation. A number of experiments have confirmed feasibility and accuracy of the optimization technique.

As a direction of future work, we would like to extend the optimization algorithms with more evolved techniques of machine learning such as deep learning.

References

1. Apache MetaModel. http://metamodel.apache.org/. Accessed Sept 2017
2. Atzeni, P., Bugiotti, F., Rossi, L.: Uniform access to NoSQL systems. Inf. Syst. **43**, 117–133 (2014)
3. Duggan, J., et al.: The BigDAWG polystore system. SIGMOD Record **44**(2), 11–16 (2015)
4. Haas, L.M.: The power behind the throne: information integration in the age of data-driven discovery. In: SIGMOD, p. 661 (2015)
5. Kolev, B., Bondiombouy, C., Valduriez, P., Jiménez-Peris, R., Pau, R., Pereira, J.: The CloudMdsQL Multistore System. In: SIGMOD, pp. 2113–2116 (2016)
6. Kraska, T., Beutel, A., Chi, E.H., Dean, J., Polyzotis, N.: The case for learned index structures. In: SIGMOD, pp. 489–504 (2018)
7. Krishnan, S., Yang, Z., Goldberg, K., Hellerstein, J.M., Stoica, I.: Learning to optimize join queries with deep reinforcement learning. CoRR, abs/1808.03196 (2018)
8. Maccioni, A., Basili, E., Torlone, R.: QUEPA: QUerying and exploring a polystore by augmentation. In: SIGMOD, pp. 2133–2136 (2016)
9. Maccioni, A., Torlone, R.: Augmented access for querying and exploring a polystore. In: ICDE, pp. 77–88 (2018)
10. Marcus, R., Papaemmanouil, O.: Deep reinforcement learning for join order enumeration. In: aiDM@SIGMOD 2018 (2018)
11. Ong, K.W., Papakonstantinou, Y., Vernoux, R.: The SQL++ semi-structured data model and query language: a capabilities survey of SQL-on-Hadoop, NoSQL and NEWSQL databases. CoRR, abs/1405.3631 (2014)
12. Fellegi, I.P., Sunter, A.B.: A theory for record linkage. J. Am. Stat. Assoc. **64**, 1183–1210 (1969)
13. Quinlan, J.R.: Simplifying decision trees. Int. J. Man-Mach. Stud. **27**(3), 221–234 (1987)
14. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann, San Francisco (1993)
15. Sadalage, P.J., Fowler, M.: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence, 1st edn. Addison-Wesley Professional, Boston (2012)
16. Stonebraker, M.: The case for polystores, July 2015. http://wp.sigmod.org/?p=1629
17. The DZone Guide To Data Persistence. https://dzone.com/guides/data-persistence-2. Accessed Sept 2017