Lachesis: Automatic Partitioning for UDF-Centric Analytics

Jia Zou
Amitabh Das
Pratik Barhate
Arizona State University
(jia.zou,adas59,pbarhate)@asu.edu

Arun Iyengar
IBM T.J.Watson Research Center
aruni@us.ibm.com

Binhang Yuan
Dimitrije Jankov
Chris Jermaine
Rice University
(by8,dj16,cmj4)@rice.edu

ABSTRACT

Partitioning is effective in avoiding expensive shuffling operations. However, it remains a significant challenge to automate this process for Big Data analytics workloads that extensively use user defined functions (UDFs), where sub-computations are hard to be reused for partitionings compared to relational applications. In addition, functional dependency that is widely utilized for partitioning selection is often unavailable in the unstructured data that is ubiquitous in UDF-centric analytics. We propose the *Lachesis* system, which represents UDF-centric workloads as workflows of analyzable and reusable sub-computations. *Lachesis* further adopts a deep reinforcement learning model to infer which sub-computations should be used to partition the underlying data. This analysis is then applied to automatically optimize the storage of the data across applications to improve the performance and users' productivity.

PVLDB Reference Format:
Jia Zou, Amitabh Das, Pratik Barhate, Arun Iyengar, Binhang Yuan, Dimitrije Jankov, and Chris Jermaine. Lachesis: Automatic Partitioning for UDF-Centric Analytics. PVLDB, 14(8): 1262-1275, 2021.
doi:10.14778/3457390.3457392

1 INTRODUCTION

Big Data analytics systems such as Spark [63], Hadoop [60], Flink [2], and TupleWare [12] have been designed and developed to address analytics on unstructured data which cannot be efficiently represented in relational schemas. By supplying user-defined functions (UDFs) written in the host language, such as Python, Java, Scala, or C++, control structures such as conditional statements and loops can be used to express complex computations. Such systems provide high flexibility and make it easy to develop complex analytics on top of unstructured data, which accounts for most of the world's data (above 80% by many estimates [54]).

Most Big Data analytics frameworks are deployed on distributed clusters and require to partition a large dataset horizontally across multiple machines [7]. Because a large dataset can be involved in multiple join-based analytics workloads, finding the optimal partitioning is a non-trivial task [1, 6, 13, 18, 28, 38, 43, 65]. Therefore, it is urgent to automate this partitioning process. Existing works in physical database design [1, 23, 28, 38, 43, 65] can well automate the partitioning for relational datasets. As illustrated in Fig. 1, they enumerate partitioner candidates based on foreign keys and select the optimal candidate using a cost-based approach.

However, it remains a significant challenge to automate this process for UDF-centric analytics workloads, as illustrated in Fig. 1. First, functional dependency that is widely utilized for partitioning selection is often unavailable in the unstructured data. Second, while the cost model based on relational algebra is widely used for selecting optimal partitioner candidates for relational applications, there is no widely acceptable cost model for UDF-centric applications due to the opaqueness of UDFs and objects [49, 67]. Third, sub-computations are opaque to the system and hard to be reused and matched for partitionings compared to relational applications.

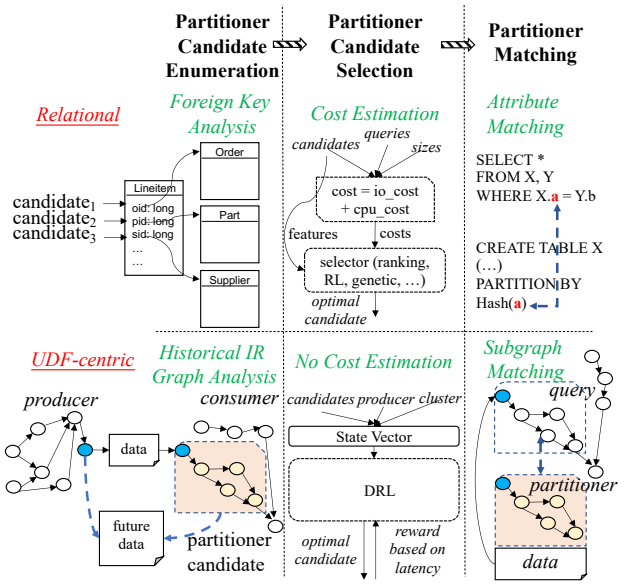


Figure 1: *Lachesis* vs. relational physical database design

Motivating Example. We have three datasets: (1) a collection of reddit *comments* objects in JSON ($\{c\}$); (2) a collection of reddit *author* objects ($\{a\}$) in CSV; and (3) a collection of the *subreddit* community objects ($\{sr\}$) also in JSON. We need to rank comments by their impacts. A classifier first predicts whether the author or the subreddit community is more important for this comment. The impact score (I_c) of a comment (c) is determined by the classification result as illustrated in Eq. 1. The classifier may use arbitrary algorithm, such as a complex deep learning neural network or a simple conditional branch such as `c.score > x`.

$$I_c = \begin{cases} \max\{a.lk, c.ck\} & if\ classify(c)\ is\ true \\ sr.ns & otherwise \end{cases} \quad (1)$$

Proceedings of the VLDB Endowment, Vol. 14, No. 8 ISSN 2150-8097.
doi:10.14778/3457390.3457392

The comment importance scores can be computed via a UDF-customized three-way join, of which the UDF that defines the join selection predicate is illustrated in Listing. 1.

Unlike SQL applications where in many cases people can simply follow the foreign keys to perform co-partitioning, UDF-centric applications may involve arbitrary logic such as the `classify()` in the above example. Even the application programmer cannot easily figure out the optimal partitioning. To make it worse, UDF-centric applications running over complex objects are almost opaque to the system, for example, the system does not understand what is happening inside the `join_selection()` function as illustrated in Listing. 1 and thus automatic enumeration, selection, and matching of partitionings for this problem become difficult.

Listing 1: UDF-centric join selection predicate

```
bool join_selection (string comment_line, string author_line, string subreddit_line) {
    string new_comment_line = schema_resolve(comment_line); //preprocessing
    json c = my_json::parse(new_comment_line); //parsing comment json object
    if (classify(c) == true) { //need to join with authors
        string c_a = c["author"];//derive author name from comment
        vector<string> r = my_csv::parse(author_line);//parsing author CSV file
        string a_name = r[1]; //derive name from author
        return (c_a == a_name);
    } else { //need to join with subreddits
        string c_sr = c["subreddit"]; //derive subreddit name from comment
        json sr = my_json::parse(subreddit_line);//parsing subreddit json object
        string sr_name = sr["name"]; //derive name from subreddit
        return (c_sr == sr_name); }}
```

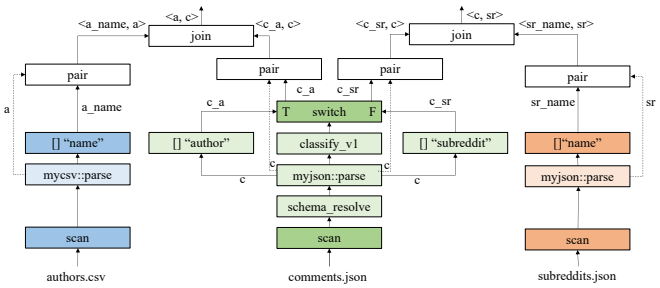


Figure 2: IR graph for Listing. 1: Partitioner candidates for authors, comments, and subreddits, are illustrated in different colors.

***Lachesis*: Automatic Partitioning.** To address the problems, we propose a data partitioning optimizer for UDF-centric workloads, called as *Lachesis* [1]. *Lachesis* allows user code to be translated into an Intermediate Representation (IR) that the system can reason. For example, the code in Listing. 1 is translated to a graph IR as shown in Fig. 2. We mainly focus on three problems in this work:

Problem 1. Partitioner Candidate Enumeration. In UDF-centric processing, a partitioner candidate can be arbitrary logic that is deeply embedded in a UDF, which is hard to identify. To address the problem, we first abstract a partitioner candidate as a two-terminal graph [5, 14, 44] that has only one unique root node (i.e., a source node that has no parents) and one unique leaf node (i.e., a target node that has no children). Then we convert the problem into a subgraph searching and merging problem. (Sec. 3.2.3 and Sec. 4.2)

Problem 2. Partitioner Candidate Selection. A UDF-based partitioner candidate may involve dynamic control flows and it is hard to predict its runtime behavior. In addition, the cost model for relational partitioner candidate selection [1, 6, 13, 18, 28, 38, 43, 65] cannot describe the overhead of manipulations (i.e, parsing, (de)compression, and (de)serialization) of arbitrary objects [4, 48]. These issues bring challenges for selecting the optimal partitioner. Therefore, we propose a deep reinforcement learning (DRL) [32, 37, 50, 51] formulation that is based on a set of unique features extracted from historical workflows for each partitioner candidate, including frequency, recency, selectivity, complexity, key distributions, number and size of co-partitioned datasets, etc.. (Sec. 3.2.2 and Sec. 4.1.3)

Problem 3. Partitioner Matching. Given a query, if its input has been partitioned using a UDF-based partitioner, the query optimizer should recognize the partitioning and decide whether a shuffling stage can be avoided. To facilitate such matching, we abstract the UDF matching problem into an IR subgraph isomorphism problem [10] utilizing the two-terminal characteristics of the partitioner IR graphs. (Sec. 3.2.3 and Sec. 4.2)

Our contributions can be summarized as:

(1) As to our knowledge, we are the first to systematically explore automatic partitioning for UDF-centric applications. We propose *Lachesis*, which is an end-to-end cross-layer system that automatically creates partitions to improve workflow performance.

(2) We propose a set of new functionalities for partitioner candidate enumeration, selection, and partitioner matching, based on subgraph searching and merging, DRL with historical workflow analysis, and isomorphic subgraph matching.

(3) We implement the *Lachesis* system and conduct detailed performance evaluation and overhead analysis.

2 BACKGROUND

2.1 IR for UDF-centric analytics

User defined function (UDF) is first proposed as an enrichment of the SQL language, to allow SQL programmers to implement their own functions for processing relational data. Later, the MapReduce and dataflow platforms such as Hadoop [60], Spark [63], Flink [2], further integrate the UDFs with high-level languages like Java, Python so that even non-relational data such as texts and images can be easily processed. However, the embedding of opaque functions that lack costing prevents UDF-centric workflows from being automatically optimized. To address the problem, several technologies were proposed in the past.

Froid [41] assumes the underlying data to be *relational*, the query that invokes the UDF must be a SQL query. Then, Froid transforms the imperative statements, conditional blocks, and loops in the UDF into relational algebraic expressions.

In contrast, other existing IRs, including Emma/Lara [3, 29], Weld [40], PlinyCompute [67], are designed for automatic optimization of UDF-centric workloads running on *unstructured data*.

Weld provides a cross-library IR based on a parallel loop operator and several declarative builders and mergers for vectors, dictionaries, and groups, to facilitate loop fusion across libraries. In Weld's implementation, a (hash) join is represented at low-level by building and probing the dictionary using the parallel loop operator. A UDF can be further represented as an abstract syntax tree (AST). Part or all of the AST tree can be replaced by the invocations of opaque C/C++ functions, depending on how much details the programmer wants to expose to the system.

[1] Lachesis is the name of a Greek god, who partitions lots and assigns fates to people. (https://en.wikipedia.org/wiki/Lachesis)

Emma/Lara and PlinyCompute provide an even more declarative IR for a k-way join operation, which can be abstracted into following expression that is similar to relational calculus:

$$\{(x_1, ..., x_k)|p(x_1, ..., x_k), x_i \in X_i, 1 \leq i \leq k\} \quad (2)$$

except that p can be represented as a UDF that processes arbitrary objects and returns a boolean value, such as the `join_selection` as illustrated in Listing. 1. Our proposed approach is designed based on top of this k-way join representation.

2.2 Storage Requirements

This work mainly considers two types of data partitioning: (1) persistent partitioning, which is to persist the partitionings of the data in the underlying storage, so that it can be reused across applications; and (2) intra-application partitioning, where the partitionings are enforced at runtime, can only live within the lifetime of an application, and is only visible to the application. Many distributed UDF-centric frameworks, such as Spark [63], support intra-application partitioning by allowing users to supply a partitioner in the application. In addition, SystemML [6], which is a linear algebra library built on top of Spark, automates the intra-application partitioning for various matrix manipulations. Intra-application partitioning is helpful for iterative joins like in PageRank, where the online repartitioning overhead can be amortized over multiple iterations in the same application. But such partitioning is inadequate for a broad class of workloads, such as data pre-processing, where a dataset is joined only once for each application. Making it worse, a non-partition-preserving operator such as `map` may easily remove an intra-application partitioning [6].

To our surprise, most popular Big Data frameworks such as Spark do not support persistent partitioning. That's because their simple storage APIs [7, 21] cannot convey the partitioning information between the storage layer and the computation layer [68, 69]. We have attempted to manually create persistent partitionings for Spark applications, but that only seems possible for Hive [53] tables via the bucketBy operator [35]. However, most UDF-centric analytics tasks cannot represent their arbitrary data in Hive tables. CoHadoop [18] allows programmers to manually enforce co-partitioning of HDFS files using MapReduce jobs and then specify this co-location relationship by changing the HDFS interface and namenode implementation, but the function used by the partitioning is hidden from the system. Therefore, the partitioning cannot be automatically reused or matched without the programmer's knowledge.

Implementation, Deployment, and Elasticity. We target at supporting both persistent and intra-application partitionings, thus we choose to implement *Lachesis* on PlinyCompute [67], which is a distributed UDF-centric analytics system implemented in C++. We use our previous work, Pangea [68], as the storage, which allows to pass the partitionings of datasets to the computation layer so that the latter can utilize such information to avoid the shuffling. *Lachesis* can be easily extended to support other distributed frameworks that compile UDF-centric workloads into analyzable IRs, and use a storage that maintains partitioning information and communicates such information to the computations.

The partitionings can be even pushed to a cloud storage like S3 that is disaggregated with the computation cluster, as long as the UDF specifying how a dataset is partitioned into multiple S3 objects is stored somewhere and queryable to the computations.

To achieve elasticity, when cluster nodes are dynamically added and removed, we can leverage existing live data migration mechanisms [7, 33] to minimize the service interruption for adding/removing nodes. In addition, we can map partitions to cluster nodes using an elastic strategy such as lazy consistent hashing, following the SnowFlake's elastic architecture design [57]. These extensions are all orthogonal to this work.

3 PROBLEM DEFINITION

In this section, we first analyze and formalize the problems. Then we summarize the challenges and main ideas.

3.1 Assumptions and Targeting Workloads

The *Lachesis* approach is based on following assumptions:
(1) The *write-once read-many assumption* that *once a dataset is written, it will be read many times*. It indicates that creating persistent partitionings while storing the data can benefit multiple workloads that take the data as input. Such a pattern is observed in a number of real-world traces [8, 61, 70].
(2) The *recurrent workflow assumption* that *a majority of workflows are re-executions* on different or incremental datasets, as widely observed in recent Microsoft and other production traces [8, 26, 27]. Therefore, we can extract partitioner candidates from historical executions of workflows and reuse these for future datasets. For example, if the *Comment-Loader* application loads a comment dataset collected in 2019 to storage; and then the *Feature-Extractor* application joins the comments dataset with the authors dataset to create feature vectors for topic recommendation, the system will think that the workflow *Comment-Loader* → *Feature-Extractor* may recur. If *Comment-Loader* later loads a new dataset collected in 2020 to storage, the input partitioning desired by the *Feature-Extractor* may be a good candidate for pre-partitioning the new dataset.

Lachesis is focused on *identifying the optimal horizontal partitioner candidates for datasets of arbitrary types*, in UDF-centric analytics workloads that involve shuffle-operations such as equi-join, group-by, and aggregations. We do not consider the vertical partitioning of arbitrary objects in *Lachesis*, because it is more complicated and requires to reason with the object layout, and thus less popular in UDF-centric systems [2, 12, 63].

Once a partitioner candidate is selected, it will be used to extract partition key(s) from each object in the dataset, and the objects that have the same keys will be dispatched to the same node. Therefore, all types of joins that can be converted into equi-join, such as array join [15] based on the equality of dimensions, attributes, or both; and similarity/fuzzy join based on the equality of locality sensitive hashing [9], can benefit from our work.

In addition, how to map keys to nodes is not a focus of this work and we use a simple hashing mechanism for that in our implementation. More advanced mapping techniques, e.g., skew-aware mapping and elastic mapping as used in array join [15, 16], and sparsity-aware recursive mapping as used in deduplication for the partitioning of band-join [31], are all orthogonal to our work, and can be incorporated to our proposed framework.

3.2 Problem Formulation

We first give an overview of the *Lachesis*' workflow. When a dataset is going to be written to storage, *Lachesis* will recommend a candidate set of IR fragments based on historical producer-consumer patterns. There are three situations: (1) If the dataset is created by a producer job that has no historical consumers, no partitioner candidates can be identified, and thus it will be partitioned using a default policy such as the round-robin policy [20]. (2) If the producer has one or more historical consumers, then, one or more partitioner candidates may be identified. Then, *Lachesis* evaluates all candidates and selects one using a deep reinforcement learning approach [51]. (3) If an existing dataset is identified to have bad organizations using certain external algorithms [24], *Lachesis* can be applied to identify the optimal partitioner candidate for reorganizing the dataset.

Lachesis focuses on two processes: (1) when a dataset is going to be stored or reorganized, the system attempts to automatically enumerate, select, and create the optimal partitioning; (2) for running applications, the system attempts to match, recognize, and utilize existing partitionings to avoid unnecessary shuffling of data. In this section, we formalize the representation of IR and partitionings in UDF-centric workflows, as well as the two processes.

3.2.1 IR and Partitioner Candidates. In this work, we define that for any workload w, there exists a mapping h that transforms w into an IR graph $a = h(w) = (V, E, S, O)$. Each node ($v \in V$) represents an atomic computation. This set of atomic computations varies with IR designs, but usually contains three categories of operators:
(1) *Lambda abstraction functions* such as a function that returns a literal (a constant numerical value or string), a member attribute or a member function from an object; unary functions such as `exp`, `log`, `sqrt`, `sin`, `cos`, `tan`, etc; or opaque unary functions if the programmer prefers not to expose the logic, such as `classify()`, `parse()`, `schema_resolve()`, etc.
(2) *Higher-order lambda composition functions* such as binary operators: `&&`, `||`, `&`, `|`, `<`,`>`, `==`, `+`, `-`, `*`, `/`, `pair`, conditional operator like `switch? on_true:on_false`; etc.
(3) *Collection-based operators* such as `scan` and `write` that reads/writes a collection of objects; `apply` that applies a lambda calculus expression (i.e., composed of lambda abstractions and higher order composition functions) to a collection of objects (like *map*); and `join`, `aggregate/fold`, `flatten`, `filter`, etc.
Each edge ($e \in E$) represents a data flow or a control flow from the source node to the destination node, as mentioned. $S \subset V$ is the set of all `scan` nodes. $O \subset V$ is a set of `write` nodes. For example, the IR derived from Listing. 1 is illustrated in Fig. 2.

As illustrated in the partitioner matching part of Fig. 1, in relational partitioning problems, a partitioner is simply a set of attributes of a relation that can be easily matched to the WHERE clause of a join query. But in UDF-centric workflows, a partitioner candidate is implicitly specified in UDFs. For example, the partitioner candidate defined in Fig. 3 is implicitly specified in the function of Listing. 1.

A partitioner candidate is represented as a two-terminal directed acyclic graph (DAG). It has only one root node that has no parents (e.g., the `scan` associated with the dataset to be partitioned) and one leaf node that has no children (e.g., the `switch` node in Fig. 3). The two-terminal graph that represents the partitioner candidate must be a subgraph in a historical consumer workload's IR graph. The leaf node in the subgraph must connect to a `pair` node for `join` in the parent graph. Given a `scan` node $s_D \in S$ that reads from the dataset $\mathcal{D}$, we can enumerate the partitioner candidates of $\mathcal{D}$ as a set of subgraphs of a consumer IR graph $a = (V, E, S, O)$, denoted as $\mathcal{F}_D$. Each $f_k = (V_k, E_k, S_k, O_k) \in \mathcal{F}_D$ satisfies $V_k \subset V, E_k \subset E, S_k = \{s_D\}$, and $\|O_k\| = 1$.

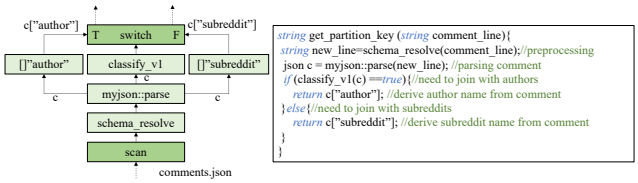


Figure 3: The 2-terminal graph that represents the partitioner candidate of the comments dataset extracted from Fig. 2

In the next two sections, we will formalize the partitioning creation and matching processes respectively.

3.2.2 Process 1. Creation of Partitionings. We first present a high-level definition of the problem, as follows. A producing workload p is going to write $\mathcal{D}$, which is a collection of n objects $\mathcal{D} = \{d_i\}, (0 \leq i < n)$, to a distributed storage C that consists of m nodes, $C = \{c_j\}, (0 \leq j < m)$. The problem is first to find a horizontal partitioning $g : \mathcal{D} \rightarrow C$, so that the overall latency of the producer p and consuming workloads of $\mathcal{D}$ is minimized, as denoted in Eq. 3. The set of l consuming workloads are represented as $\mathcal{W} = \{w_k\}, (0 \leq k < l)$, lat_p represents the latency of the producer, and $freq_k$ and lat_k denote the execution frequency and latency of w_k respectively. Then the selected partitioning g_{opt} needs to be automatically applied while storing $\mathcal{D}$ to the cluster C.

$$g_{opt} = \arg \min_{g:\mathcal{D}\rightarrow C} (lat_p + \sum_{\forall w_k \in \mathcal{W}} (freq_k \times lat_k)) \quad (3)$$

We further formulate a more detailed model by lowering down the partitioning functions (g). There exist m^n different partitioning functions, to prune which, we only consider well-known partition strategies such as hash partitioning, range partitioning, round robin partitioning, and random partitioning [66].

A hash partitioner is defined by a function $f_{keyProj}$ that extracts the partition key from a data item, where the key must have a hash function defined. For this type of partitioner, given $f_{keyProj}$, the corresponding g is defined as $g_{hh}^{f_{keyProj}}(d_i) = hash\,(f_{keyProj}(d_i))\%m$, $\forall d_i \in \mathcal{D}$. Range partitioners are similar, except that the partition key must have a comparator defined for sorting; and g is accordingly defined as $g_{rn}^{f_{keyProj}}(d_i) = range(f_{keyProj}(d_i))\%m$. Round robin and random partitionings do not require any functions. The former is defined as $g_{rr}(d_i) = next\_int()\%m, \forall d_i \in \mathcal{D}$, and the latter is denoted as $g_{rm}(d_i) = random()\%m$. Therefore, given a set of q different $f_{keyProj}$ for partitioning the dataset D, denoted as $\mathcal{F} = \{f_i\}, 0 \leq i < q$, the search space includes all $2q$ combinations of the two partition strategies (i.e. hash or range) and the q functions, plus the round robin and random strategies, represented as

$\mathcal{G}^{\mathcal{F}} = \{g_{hh}^{f_0}, ..., g_{hh}^{f_{q-1}}\} \cup \{g_{rn}^{f_0}, ..., g_{rn}^{f_{q-1}}\} \cup \{g_{rr}, g_{rm}\}$. Thus Eq. 3 can be lowered into Eq. 4:

$$g_{opt} = \arg\min_{g \in \mathcal{G}^{\mathcal{F}}} (lat_p + \sum_{\forall w_k \in \mathcal{W}} (freq_k \times lat_k)) \quad (4)$$

Latency is influenced by numerous factors such as CPU costs, I/O costs, hardware parallelism, memory size, network bandwidth. Because of the lack of a widely acceptable cost model for UDF-centric analytics, instead of detailing all of these factors, we choose to use a DRL approach to optimize the objective purely based on the observed latency (used to compute reward) and features regarding each partitioner candidate as well as the data and the environment, which we will describe in detail in Sec. 4.1.3.

3.2.3 Process 2. Match of Partitionings. Supposing the comments dataset is partitioned by the candidate illustrated in Fig. 3, when the *Feature-Extractor* workload (Fig. 2) processes the data, the system should recognize that the partitioner assoicated with the data is a desired partitioning of this workload. If the subreddits and authors datasets are also co-partitioned for the workload, the query optimizer will schedule local joins to avoid the shuffling stages.

A partitioner candidate, except for the random or the round robin partitioners, is a pair of $f_{keyProj}$ and its partition strategy (hash or range). Supposing a partitioner candidate, with its IR represented as $f_D = (V_D, E_D, S_D, O_D)$, has been applied to a dataset $\mathcal{D}$. Then if an application $w \in \mathcal{W}$ reads from $\mathcal{D}$ and we have $a = h(w) = (V, E, S, O)$ as the IR graph of w, there must exist a `scan` node $s_D \in S$ that reads from $\mathcal{D}$, denoted as $s_D = a.find\_scanner(\mathcal{D})$. In addition, if there exists a subgraph of a, which is equivalent to f_D, the system's query scheduler can simply avoid the execution of this subgraph, because the partitioning represented by this subgraph has already been applied to $\mathcal{D}$. The identification of such subgraphs can be abstracted into a subgraph isomorphism problem [10] : *Given two graphs $f_D = (V_D, E_D, S_D, O_D)$ and $a = (V, E, S, O)$, a subgraph isomorphism from f_D to a is to find a function $f : V_D \rightarrow V$ such that if $(u, v) \in E_D$, then $(f(u), f(v)) \in E$ and if $s \in S_D$, then $f(s) \in S$.*

3.3 Summary of Challenges

UDF-centric applications are very different with relational applications where costs of a query execution plan are easy to estimate, and a partitioning predicate can be easily extracted (i.e., searching in WHERE clause) and reused (i.e., appending a PARTITION BY predicate). The specific challenges include:

(1) Workload Enumeration. Given an incoming/existing dataset $\mathcal{D}$, how to obtain the set of consuming workloads $\mathcal{W}$?

(2) Enumeration of Partitioner Candidates. How to obtain the set of partitioner candidates $\mathcal{F}$ for partitioning a dataset $\mathcal{D}$?

(3) Optimization. How to solve the optimization problem illustrated in Eq. 4, with the lack of a widely acceptable cost model?

(4) Match of Partitionings. How to efficiently solve the subgraph isomorphism problem, which is NP-complete [10]?

These challenges are addressed in *Lachesis* based on following ideas:

(1) We utilize historical workflow execution information to predict future workloads based on the workload recurrence patterns.

(2) Partitioner candidates or existing/desired partitionings are just a special type of subgraphs. Recognizing such subgraphs in an IR may be simpler than the general subgraph isomorphism problem.

(3) A DRL-based approach that models the dynamic factors purely through rewards of past decisions may solve the optimization problem with good adaptivity and also avoid the costs of profiling the hardware environments as required in a cost model approach.

4 OUR SOLUTIONS

4.1 Creation of Partitionings

4.1.1 Workload Enumeration. Given a producer p that is going to write a dataset $\mathcal{D}$ to the storage, and a set of nw historical workloads $\mathcal{W}' = \{w_i'\}, (0 \leq i < nw)$, how to enumerate the set of workloads $\mathcal{W}$ that may process $\mathcal{D}$ in the future?

Based on the *recurrent workflow assumption*, if there exists $w_i' \in \mathcal{W}'$ with $h(w_i') = (V_i', E_i', S_i', O_i')$ that is isomorphic to p with $h(p) = (V, E, S, O)$, which means an isomorphism bijection $f : h(p) \rightarrow h(w_i')$ exists, then for $o_D \in V$ that is the node outputting $\mathcal{D}$ in $h(p)$, there must exist $o_D' \in O_i'$ so that $f(o_D) = o_D'$. Further more, if $\exists\, w_j' \in \mathcal{W}'$ with $h(w_j') = (V_j', E_j', S_j', O_j')$, $s_D \in S_j'$, satisfying that the dataset read by s_D is created by o_D', we can conclude that w_j' once consumed the output of w_i', so it may consume the output of p in the future (because of the isomorphism between $h(p)$ and $h(w_i')$), and thus we have $w_j' \in \mathcal{W}$.

We encapsulate the above process into a historical workflow analysis component. It first reconstructs low-level workflow information from execution logs, which is illustrated in Fig. 4(a), where each node represents an execution of a workload, identified by (app_id, timestamp) and each edge represents a historical dataset created by its source node, and consumed by its destination node. It then further condenses the low-level graph into a skeleton graph [45, 58] by merging nodes that have the same IRs and thus expect exactly the same partitionings, as illustrated in Fig. 4(b). In the skeleton graph, each edge represents a list of historical execution runs in the form of (app_id, timestamp, input_data_id, output_data_id). Given a currently running application belonging to group1 that is going to write a dataset to the storage, based on the skeleton graph in Fig. 4(b), *Lachesis* will predict that applications from group2 and group4 may process the dataset in the future. The matching of $h(w_i')$ to $h(p)$ is achieved by offline computing a hash signature for each workload's IR graph ($h(w_i')$) through enumerating, sorting, and concatenating all distinct paths that connect a `scan` node to a `write` node [55]. These signatures are stored into a hash table and then the lightweight online process matches the signature of the producer's IR graph ($h(p)$) against the hash table.

4.1.2 Partitioner Candidate Enumeration. Given a set of consuming workloads $\mathcal{W}$ enumerated for $\mathcal{D}$, $\forall w_i \in \mathcal{W}(0 \leq i < nw)$, we can further enumerate a set of partitioner candidates, with each being a subgraph, denoted as $a_i = h(w_i) = (V, E, S, O)$.

As mentioned in Sec. 3.2.1, the subgraph representing a partitioner candidate must satisfy that the sole root node is a `scan` node ($s_D \in S$) that reads from $\mathcal{D}$, and the unique leaf node is a node that connects to a `pair` node followed by a `join` node. Obviously satisfying these conditions makes the subgraph sufficient to serve as a partitioner candidate. To efficiently identify such subgraphs, we propose a two step approach, as illustrated in Fig. 5. The first step is to recursively traverse a_i and enumerate all distinct paths that start at the `scan` node s_D and end at

any of the pair→join paths. We formalize this process in Alg. 1. The second step is to merge all paths that connect the same scan node and the same leaf node into one graph to serve as one partitioner candidate, as illustrated in Alg. 2. Thus we can formalize the process of enumerating all partitioner candidates from $\mathcal{W}$: $\hat{h}_{\mathcal{W}\to\mathcal{F}} = \bigcup_{w_i \in W}\{merge(search(h(w_i)), \mathcal{D}, \emptyset)\}$.

Algorithm 1 $search(a_i, s_D, F_i)$

INPUT1: $a_i = (V, E, S, O)$ (the IR graph of one of $\mathcal{D}$'s consuming workloads $w_i \in \mathcal{W}$)
INPUT2: s_D (the scan node in a_i that reads from $\mathcal{D}$)
INPUT3 and OUTPUT: F_i (a list of partial partitioner candidates for $\mathcal{D}$ extracted from a)
$F_i \leftarrow \phi$
$V' \leftarrow \{s_D\}; E' \leftarrow \emptyset; S' \leftarrow \{s_D\}; O' \leftarrow \emptyset$
for v_k in $s_D.children$ **do**
  $V' \leftarrow V' \cup \{v_k\}$
  $E' \leftarrow E' \cup \{edge(s_D, v_k)\}$
  $O' \leftarrow \{v_k\}$
  if $\nexists path(v_k, pair, join)$ **then**
    $F^t \leftarrow \emptyset$
    $search((V - V') \cup \{v_k\}, E - E', (S - S') \cup \{v_k\}, O'), v_k, F^t)$
    for $f^t = (V^t, E^t, S^t, O^t) \in F^t$ **do**
      $F_i \leftarrow F_i \cup \{(V' \cup V^t, E' \cup E^t), S', O^t\}$
    end for
  else
    if $E' \neq \emptyset$ **then**
      $F_i \leftarrow F_i \cup \{(V', E', S', O')\}$
    end if
  end if
end for
return F_i

Algorithm 2 $merge(F_i)$

INPUT: F_i (a list of partial partitioner candidates output from Alg. 1)
OUTPUT: F'_i (a list of partitioner candidates)
$hashmap \leftarrow \emptyset$
$F'_i \leftarrow \emptyset$
for $f_k = (V_k, E_k, S_k, O_k) \in F_i$ **do**
  if $hashmap.count((S_k, O_k)) \neq 0$ **then**
    $(V^t, E^t, S^t, O^t) \leftarrow hashmap[(S_k, O_k)]$
    $hashmap[(S_k, O_k)] \leftarrow (V^t \cup V_k, E^t \cup E_k, S_k, O_k)$
  else
    $hashmap[(S_k, O_k)] \leftarrow f_k$
  end if
end for
for $((S_k, O_k), f_k) \in hashmap$ **do**
  $F'_i = F'_i \cup \{f_k\}$
end for
return F'_i

4.1.3 DRL-based Optimization. Once a set of partitioner candidates are enumerated, the next step is to select the optimal one to apply. There are existing works targeting similar data partitioning optimization problems in OLAP relational databases [1, 17, 22, 23, 34, 38, 43, 47, 64]. These works, including recent RL-based partitioning advisors [22, 23], are largely depending on the functional dependency [42] and cost models of relational databases, which may not exist in UDF-centric analytics. In this work, we choose a DRL approach based on the actor-critic network [50], which integrates the best of both worlds of value-based (e.g., Q-Learning [19, 59]) and policy-based (e.g., proximal policy optimization [46]) RL. We use A3C algorithm [37], which is a state-of-the-art algorithm for learning the actor-critic network [36]. It also allows to use multiple learning agents to accelerate the training process.

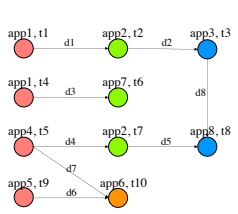
The actor-critic network is based on policy gradient. It takes a *state* vector, which describes the environment, as input, and outputs *policy*, which is a probability distribution in the *action* space. Then, the critic network also takes *state* as input, and outputs the expectation of value function that will be used together with *reward* to compute the policy gradient to improve the learning for both of the actor and critic networks. The optimization goal of the model is to minimize the cumulative processing latency of current and future applications. We formulate the DRL problem in detail as follows.

State. To formualte the *state* feature vector, we first consider following features for each of the k-most recent partitioner candidates:

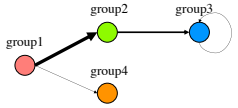
1. frequency indicates the total number of historical executions of the IR where the partitioner candidate is extracted from.
2. distance indicates the average time interval between the most recent two runs in the candidate's IR group mentioned in Fig. 4(b).
3. recency indicates the timestamp of the most recent run of applications in the candidate's IR group.
4. complexity computes the number of nodes in the subgraph that represents the partitioner candidate.
5. selectivity indicates the ratio of the average size of the partition keys extracted to the average size of source objects. This metric measures the amount of data that should be shuffled at runtime if this partitioner candidate is desired but not selected.
6. key_distribution indicates the average number of unique values generated by hashing the output of the partitioner candidates in historical runs. The key distribution affects the system load balance. If the output keys are skewed, most of the objects may be stored on the same worker instance, while only a small portion of objects are distributed in other workers.
7. num_copartitioned indicates the number of existing datasets that will be co-partitioned with the data if this partitioner candidate is selected. These datasets are identified by searching their partitionings in the IR where this partitioner candidate's IR is extracted.
8. size_copartitioned indicates the total sizes of the co-partitioned datasets mentioned above.

We compute the pearson correlation coefficient (PCC) [30], which is a measure of the linear correlation coefficient between two random variables, for the reward (which we will describe later) and each of aforementioned features. The results show that frequency, num_copartitioned, size_copartitioned are the top three features that are mostly correlated with the reward. In addition, the PCC of recency and distance to the reward will increase with the temporal locality of the workload patterns. While the rest of the features have significantly less PCC with reward, they are also useful in avoiding some bad partitioner candidates, e.g., an aggregation/join key extraction function that maps all elements into a few keys with skewed distribution by using key_distribution.

Besides the features that describe each of top k partitioner candidates, the other features we use include the estimated size of the



(a) Low-level Graph



(b) Skeleton Graph

Figure 4: Workflow representation (each node is a workload IR graph)

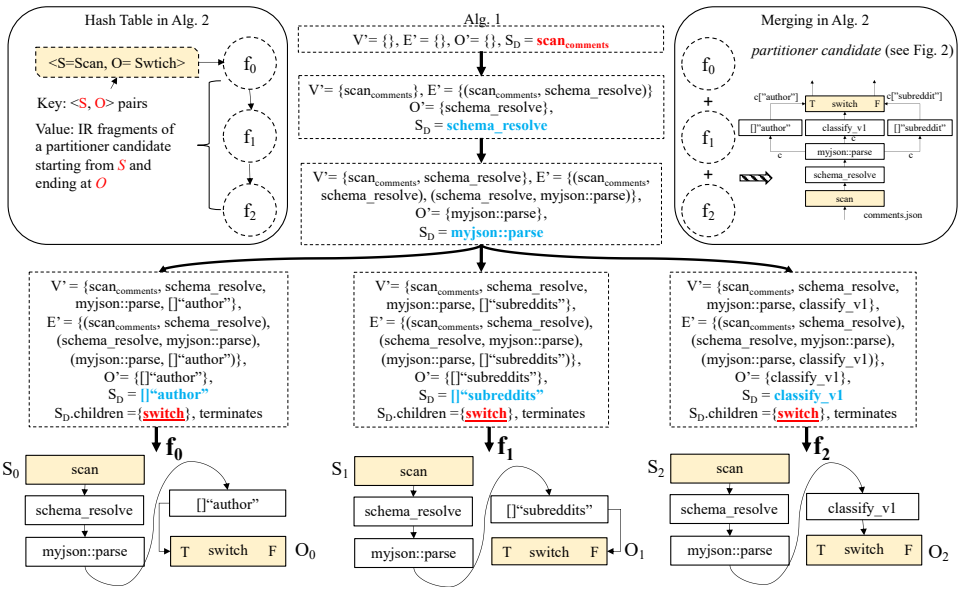


Figure 5: Running example for Alg. 1 and Alg. 2

dataset to be dispatched, the number of workers, the number of cores and sizes of available memory and disk space on each worker. All features are normalized before being used.

Action Space and Policy. Upon receiving the *state* vector s_t, the RL agent needs to send back an action a_t that corresponds to the selected partitioner candidate. The RL agent selects actions based on a policy, defined as a probability distribution over candidate lambdas: $\pi = \pi_\theta\{s_t, a_t\} \rightarrow [0, 1]$. Here θ is the hidden parameter that controls the policy, which is represented by the actor neural network [37]. The action space can be extended to sample and select more than one actions, for creating multiple replicas, with each organized using a different partitioning [68, 69].

Reward Function. *Lachesis* also needs to compute reward r_{t-1} for last action a_{t-1}. Because latency will increase with data size, we define the reward function to be the performance speedup of the total throughput of applications that consume the dataset for which action a_{t-1} is applied, compared to a baseline throughput. The baseline is the average throughput of the historical executions of these applications. The reward function is formalized as below. W^t_{t-1} represents all workloads that have processed the dataset partitioned at time $t-1$, during the period from time $t-1$ to t. W' represents all historical workloads used for workload enumeration.

$$r_{t-1} = \frac{\sum_{w \in W^t_{t-1}} \sum_{\mathcal{D} \in w.input} size(\mathcal{D}) / \sum_{w \in W^t_{t-1}} latency(w)}{\sum_{w' \in W'} \sum_{\mathcal{D} \in w'.input} size(\mathcal{D}) / \sum_{w' \in W'} latency(w')}$$

Policy Gradient. Policy gradient methods estimate the gradient of the expected total reward by computing the gradient of cumulative discounted reward with respect to the policy, which can be represented as [37]: $\nabla_\theta E[\sum_{t \geq 0} \gamma^t r_t | \pi_\theta] = E[\nabla_\theta log \pi_\theta(s, a) A^\theta(s, a) | \pi_\theta]$. $A^\theta(s, a)$ is called advantage function that indicates how much better an action is compared to the expected. Each update of the *actor network* follows the policy gradient to reinforce actions that lead to better rewards: $\theta \leftarrow \theta + \alpha \nabla_\theta log \pi_\theta(s_t, a_t) A(s_t, a_t) + \beta \nabla_\theta H(\cdot | s_t)$.

Here, α is the learning rate; $H(\cdot)$ is the entropy of the policy, which is to encourage exploration in the action space; and β is used to control the emphasis in exploration over exploitation.

To compute the advantage function $A(s_t, a_t)$, we need estimate the value function $V^{\pi_\theta}(s)$ as $Q(s, a)$. The *critic network* is responsible to learn the estimate of the value function from observed rewards. All the details of derivation can be found in reference [37].

An End-to-End Algorithm. Based on the proposed workflow enumeration, partitioner candidates enumeration, and optimization, we give Alg. 3 to describe the end-to-end partitioning creation process.

Algorithm 3 *partitioning_creation*$(p, \mathcal{D}, \mathcal{W}')$

```
INPUT1: p (the producer workload)
INPUT2: D (the dataset to store and partition)
INPUT3: W' (the set of historical workloads)
W ← match(p, W') {Sec. 4.1.1}
F ← ∅
for w_i ∈ W do
    a_i ← h(w_i) {via DSL/IR: Sec. 3.2.1}
    s_D ← a_i.find_scanner(D){Sec. 3.2.3}
    F_i ← merge(search(a_i, s_D, ∅)) {Sec. 4.1.2: Alg. 1 and Alg. 2}
    F ← F ∪ F_i
end for
g_opt ← min_{g∈G^F}(lat_p + Σ_{∀w_k∈W} (freq_k × lat_k)){Sec. 4.1.3}
for ∀d_i ∈ D do
    store d_i to the node g_opt(d_i)
end for
```

4.2 Matching of Partitionings

In this section, we discuss how to match the partitioning of an input dataset to the running application (Process 2). While subgraph isomorphism problem is NP-complete [10], we can utilize the two-terminal DAG characteristics of the subgraph associated with a partitioner candidate to provide an efficient solution.

Given a dataset $\mathcal{D}$, we can obtain the IR graph of its partitioning through the storage interface, denoted as $f_D = (V_D, E_D, S_D = \{s\}, O_D = \{o\})$. Also given a running application w that takes $\mathcal{D}$ as

one of its inputs, we can obtain w's IR graph $a = h(w) = (V, E, S, O)$. We first locate the scanner node $s_D \in S$ that connects to $\mathcal{D}$, and create an isomorphic mapping from s to s_D, because these two are root nodes and they must match with each other due to the uniqueness of root node. Then we recursively visit each descendant (denoted as v) of s_D in IR DAG a (we use depth-first search for this step). Each time meeting a v that is a `pair->join` path, we create a candidate isomorphic subgraph $IG^{(s_D,v)}$ that connects the root node s_D and the leaf node v. This step can be accelerated by indexing all `join` nodes when constructing the IR graph. Then for each candidate isomorphic subgraph, we create a signature for each distinct path from s_D to v by concatenating the node labels along the path. We can thus derive a unique signature to identify each subgraph by further concatenating all path signatures sorted in lexicographical order. By matching the signatures of the f_D and each candidate subgraph, we can find all isomorphic subgraphs. The algorithm is illustrated in Alg. 4. It can be further optimized by using a hashmap to store the signatures of candidate subgraphs.

Algorithm 4 $partitioning\_match(f_D, ssset_D, a, P)$

```
INPUT1: f_D = (V_D, E_D, S_D = {s}, O_D = {o}) (IR of the dataset's partitioning)
INPUT2: ssset_D (sorted set of signatures for all paths in f_D)
INPUT3: a = (V, E, S, O) (IR of the running consumer workload)
INPUT4: P = {p_i} ⊂ V (the set of pair->join paths)
OUTPUT: I (the set of subgraphs in a that is isomorphic to f_D)
I ← ϕ
s_D = a.find_scanner(D)
for p_i ∈ P do
    path_set = a.find_all_paths(s_D, p_i)
    if path_set ≠ ϕ then
        sig_set ← ϕ
        for path ∈ path_set do
            sig ← create_signature(path)
            sig_set ← sig_set ∪ {sig}
        end for
    end if
    if ssset_D equals to sorted(sig_set) then
        I ← I ∪ {IG^(s_D,p_i)}
    end if
end for
return I
```

5 SYSTEM IMPLEMENTATION

We implement *Lachesis* on top of a baseline system, PlinyCompute [67], which is a UDF-centric analytics framework, using the Pangea storage [68][2]. *Lachesis* stores information regarding historical application executions, including the paths, sizes, partitionings of datasets, IRs, runtime statistics such as execution latency, output data sizes for each job stage, in a SQLite database. The producer-consumer relationships among applications are reconstructed to provide a full picture of historical workflow executions. Given a producer that materializes a dataset, the historical workflow analyzer can efficiently supply a set of applications that once have processed the datasets created by the same type of producers. Based on the *recurrent workflow assumption*, each application in this set may re-process the dataset in the future. Therefore, any relevant partitioning computations extracted from these applications may be considered as a partitioner candidate for this dataset. The historical workflow analyzer is also responsible for selecting and extracting features for the top k partitioner candidates.

DRL Model Training. The RL model is deployed using TensorFlow. Ideally, the training would occur with actual data loading and workload execution. However, this will be slow because to compute the *reward*, the RL client needs to wait for all of the related datasets to be loaded using the partition scheme specified in recent *action*s and related queries to be executed. To avoid this overhead, existing DRL approach for relational data partitioning chooses to bootstrap the model using traces generated by a cost model [22]. While this is a reasonable solution for relational database, there is no widely-accepted cost models for UDF-centric analytics [49].

To alleviate the training overhead, we propose to transform the running statistics of a few actual query executions into the estimated statistics of a large set of diversified workloads randomly generated from the queries. We first select a few queries. (One requirement is that some of these queries should be latency-sensitive to the partitionings of their input data, e.g., queries involving join and aggregation.) Then, we enumerate all partitioner candidates for the inputs of these queries. Furthermore, we run the queries and measure each query's latency for each possible partition scheme.

In each iteration, the training component generates a workload by randomly combining the pre-executed queries with varying frequencies, and it forms the *state* simply based on the historical statistics of all possible partitioner candidates for the queries in the workload. Then it sends the *state* to the RL server and obtains the action for partitioning. Instead of actually partitioning the data and running the queries from the generated workload, it simply looks up the latency related to this action and computes reward from historical latency statistics of these queries.

In this way, we can generate an unlimited number of workloads for training, and we do not need to actually run any queries of the workload. The increased number of workloads result in more robust representations of important features such as frequency, recency, number of co-partitioned datasets, etc., as mentioned in Sec. 4.1.3.

6 EVALUATION

In this section, we mainly want to answer following questions:
(1) What are the performance gains that can be achieved by *Lachesis*'s automatic persistent partitioning for different types of Big Data analytics applications? (Sec. 6.2, 6.3, 6.4)
(2) How much online and offline overhead is incurred during the automatic partitioning process? (Sec. 6.5)
(3) How effective is the DRL training process and how much time and efforts are required for training? (Sec. 6.6)

6.1 Environment Setup

6.1.1 Workloads. To answer the questions, we implement a set of representative workloads including:
(1) **Reddit data integration workflows.** We implement two dynamic workflows: one is the motivating example illustrated in Fig. 2; the other is a deep learning model inference workflow.

[2]The *Lachesis* code is available: https://github.com/asu-cactus/lachesis

(2) **TPC-H Queries.** We implement ten TPC-H queries (Q1, Q2, Q3, Q4, Q6, Q12, Q13, Q14, Q17, Q22). Each table is represented as a collection of C++ objects and each predicate is wrapped as a UDF. The producer workloads load eight TPC-H datasets (lineitems, orders, customers, parts, suppliers, partsupp, regions, and nations), to the storage, and then the queries run to process the loaded data.
(3) **PageRank Analytics workflow.** We implement a web analytics workflow that consists of two workloads: pre-processing the web pages, and running PageRank iterations on the pages [39].

6.1.2 Baselines. For the TPC-H benchmarks, we compare *Lachesis* to the partition schemes suggested by a commercial distributed database. For other workloads, because there are no existing automatic partitioners that can work with UDF-centric workloads to process arbitrary data types, we measure the performance speedup by comparing the consuming workload's latency of applying *Lachesis* to different baselines listed as follows.
(1) Heuristics that are typically used by a database administrator [23, 64]: one is to co-partition all datasets with the most frequent joined dataset (i.e. Heuristics(a)) and the other is to co-partition all datasets with the largest dimension table (i.e. Heuristics(b)).
(2) The round robin dispatching strategy, which is to dispatch each page of data to a cluster node in order. This is an effective way to guarantee load balance for large dataset, and is adopted by many storage systems such as IBM GPFS [20].
(3) For the Reddit workflow-1 and the PageRank workflow, we also compare *Lachesis* to a reactive approach [24].

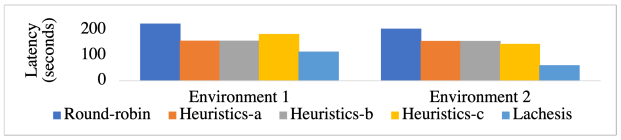
6.1.3 Environment Setup. We use three AWS clusters; (1) Environment 1, which is a cluster that has three r4.2xlarge instances. Each r4.2xlarge instance has 8 CPU cores, 61GB memory, up to 10 GB network connection. (2) Environment 2, which has eleven r4.2xlarge instances, up to 10 GB network connection. (3) Environment 3, which has six m2.4xlarge instances. Each instance has 8 CPU cores, 68 GB memory, up to 1 GB network connection. In each cluster, one instance serves as the master and the rest of the instances serve as workers. Each instance uses 200GB Elastic Block Store SSD.

6.2 Reddit Data Integration Workflows

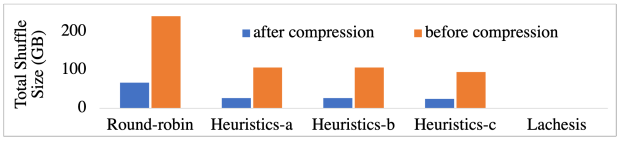
We compare the performance of two Reddit data integration workflows by using different partitioning strategies. In each workflow, three workloads are responsible for loading the Reddit comments in JSON format (up to 100 millions of comments objects, 128 gigabytes in total), Reddit authors data in CSV format (78 millions of authors objects, 10 gigabytes in total), and subreddits data in CSV format (3.7 gigabytes in total).[3]

We store the raw files in a S3 bucket, and the time to copy all the files from S3 to the master node of the cluster is 76 seconds. Such loading happens only once and the loaded data will be repeatedly processed and the following measurements do not include this time.

6.2.1 Workflow-1: three-way dynamic join. For this experiment, we use 10% of data from the three Reddit datasets in Environment 1; and use all data in Environment 2. After loading the data to storage, the workflow performs a three-way join similar to Listing. 1, depending on a classifier that simply checks whether the value of



(a) speedup



(b) shuffled bytes

Figure 6: Reddit Workflow-1

a comment's score is larger than a constant. In this workflow, the classifier results in around 50% of comments to join with authors, and 50% of comments to join with subreddits.

Because the workflow contains only one three-way join, both heuristics-a and b choose to partition the comment objects based on author name. Existing physical database design advisors enumerate partition keys based on foreign keys. These tools will either choose to partition the Reddit comments dataset along the author name, or along the subreddit channel. So we added heuristics-c to partition along the subreddit channel for completeness. Only our *Lachesis* approach can identify and exploit the UDF-based partitioner candidate as illustrated in Fig. 3. As a result, *Lachesis* achieves 1.4× speedup in the two-worker cluster and 2.4× speedup in the ten-worker cluster, as shown in Fig. 6(a). The total amount of data shuffled in the Environment 2 is illustrated in Fig. 6(b). Furthermore, we observe that with the round-robin partitioning, in Environment 2, in average, it needs to shuffle 32 gigabytes to co-locate with the authors data; and 29 gigabytes to shuffle for co-locating with the subreddits data, after compression using Snappy v1.5.

As mentioned at the beginning of Sec. 3.2, *Lachesis*' functionalities for enumerating and matching partitioner candidates can be used to repartition existing datasets based on new incoming queries (i.e., reactive approach). We thus design several scenarios that are composed of repeated executions of two queries: Q1, a two-way join of reddit authors and comments; and Q2, the three-way dynamic join that we just discussed. The results are illustrated in Fig. 7. When Q1 is repeatedly executed, the reactive approach chooses to co-partition the comments and authors datasets by the UDF that parses the author name; Then when Q2 is repeatedly executed, the reactive approach chooses to further partition a subset of the comments dataset by the UDF that parses the subreddit name. If the queries are executing in a pattern of Q1, Q2, Q1, Q2, ..., the reactive approach only chooses to repartition once based on the author-extraction UDF. *Lachesis* and Heuristics-a/b always choose to pre-partition the comments and authors datasets based on the author-extraction UDF in all of these patterns. We can see that if a query execution pattern repeatedly occurs, the *Lachesis-Reactive* approach can achieve similar or slightly better performance.

[3]Reddit datasets are download from http://files.pushshift.io/reddit/.

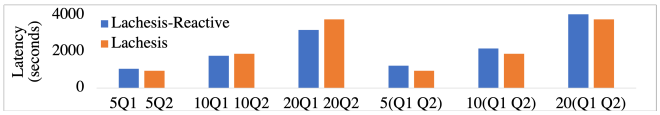


Figure 7: Comparison of Lachesis-reactive and Lachesis in Environment 1. kQ1 kQ2 is to execute Q1 for k times and then Q2 for k times. k(Q1Q2) is to repeat the execution of Q1 and Q2 for k times.

6.2.2 Workflow-2: Reddit Comment Classification based on Deep Neural Network (DNN). We implement a DNN model serving workflow[4] that classifies whether a Reddit comment should join with author info or subreddit channel info, as mentioned in the motivating example in Sec. 1. Considering that features are extracted from all the attributes regarding the comment object, occurrences of words in a large dictionary, and ngrams, this is a high-dimensional machine learning problem.

This workflow consists of six jobs: (1) blocking, which is to extract a pre-computed feature vector from a comment object through an index, and block the feature vectors of a batch of comments into many 2-dimensional 1000 × 1000 matrix blocks; (2) layer-1, which passes the feature blocks to the first fully-connected layer that has 1000 neurons and a 1000-dimensional bias vector with Relu activation, and outputs y_1; (3) layer-2, which passes y_1 to the second fully-connected layer that has 2000 neurons and a 2000-dimensional bias vector with Relu activation, and outputs y_2; (4) layer-3, which is the output layer that consists of 2 neurons for the two labels and a 2-dimensional bias vector, and outputs y_o; (5) flattening, which applies softmax activiation to y_o to get the probability distribution over the labels for each comment, and flattens result tensor blocks to a set of comment label objects; (6) labeling, which joins the batch of comments with the label objects, so that each comment's label attribute is filled with the prediction.

We run the experiment with 200000 to 1 million features in Environment 1, using a batch size of 1000. All values are using double precision. We observe that in this case, by only partitioning the persistent datasets such as reddit comments, the weight and bias matrices of layer-1, layer-2, and layer-3, it achieves only moderate performance gain, labeled as *Lachesis-persistent* in Fig. 8. By additionally co-partitioning intermediate data such as the output of the blocking, layer-1, layer-2, layer-3, and flattening, we can achieve the maximum performance gain, labeled as *Lachesis-full* in Fig. 8.

Although this workflow has more than ten datasets that are inputs to join operations, for each dataset, usually only one or two partitioner candidates exist. So the *Lachesis-full* result is similar to Heuristics-a and Heuristics-b. Given the many datasets involved and the complexity of the workflow, it is hard for programmers to manually figure out and manage the partitionings. The productivity brought by automatic partitioning is a great benefit of *Lachesis*.

A significant portion of overall performance gain (1.3× to 1.6× speedup) is coming from the blocking stage and the layer-1 stage. The blocking stage transforms and aggregates the batch of comments that is merely 1 megabytes in total to a set of feature blocks that have 2GB to 10GB in total size depending on the number of features. Because *Lachesis* chooses to partition comments by its unique identifier, comments are evenly distributed across worker threads. However, the round-robin approach distributes data by

[4]The model architecture is consistent with the simple FFN example in TensorFlow [52], but implemented in PlinyCompute [67] using Tensor Relational Algebra [25, 62].

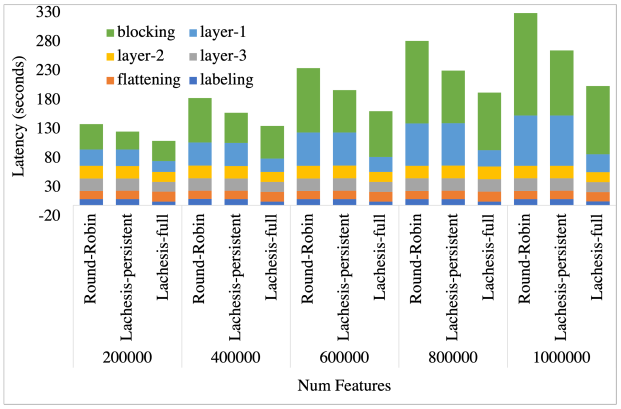


Figure 8: Performance of Workflow 2 for a batch of inferences

pages, which causes skewed distributions due to the small size of a batch. The layer-1 stage includes a join of a *numNeurons*×*numFeatures* weight matrix and a *numFeatures*×*batchSize* input matrix, which can benefit from *Lachesis* by avoiding shuffling. Without *Lachesis*, for 1 million features, we observed about 8.5GB gets shuffled (5.6GB w/ compression), in Environment 1. Most of the shuffling gets avoided with Lachesis-full.

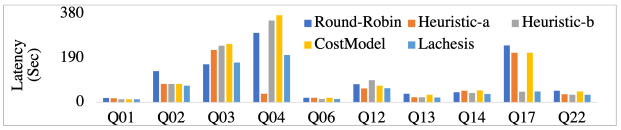
The latency of the layer-2, layer-3, flattening, and labeling will not change with the increase in number of features because the joins in these stages are determined by factors such as the number of neurons and the batch size. Despite of small shuffle sizes in these stages, *Lachesis* can still achieve about 15% performance gain because the removal of the shuffling phases reduces the CPU costs.

6.3 TPC-H Refactored with Objects and UDFs

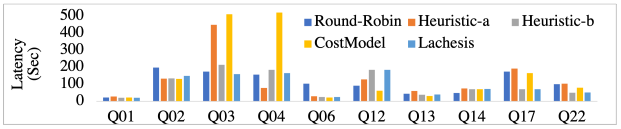
We implement all eight TPC-H tables as eight C++ classes. Then we implement ten TPC-H queries that involve aggregations and/or joins and can be represented using PlinyCompute computations. We load data that is generated using dbgen at scale *SF-10* into a Environment 1; and load data at scale *SF-100* into Environment 2. For this experiment, we compare to round-robin (RR) partitioning, Heuristics-a, Heuristics-b, and the partitionings automatically created by a commercial distributed database using a cost-based physical database design advisor (denoted as CostModel). The selected partitionings and the total execution latency of ten queries for each partitioning strategy are shown in Tab. 1 and Tab. 2. The measured latency for each query is illustrated in Fig. 9.

In both environments, *Lachesis* achieves the best performance. In Environment 1, it outperforms the second best strategy, which is Heuristic-a, by 12%. In Environment 2, *Lachesis* outperforms Heuristic-b, which is the second best, by 6%. The CostModel approach shows the worst performance in both enviornments, which indicates that the cost model of a relational database system is not applicable to a UDF-centric analytics system that involves more complicated overhead for manipulating arbitrary objects.

Taking Q17 for example, in both environments, *Lachesis* chooses to co-partition the lineitem table and the part table on partkey. As a result, for this query, *Lachesis* achieves 5× speedup in Environment 1 and 3× speedup in Environment 2, compared to Hueristics-a. We also observe that for Q04, Hueristics-a significantly outperforms



(a) Environment 1: SF-10



(b) Environment 2: SF-100

Figure 9: TPC-H performance (refactored with objects and UDFs)

Table 1: Comparisons of TPC-H Partitionings in Environment 2

	RR	Heuristics-a	Heuristics-b	CostModel	Lachesis
customer	-	c_custkey	c_custkey	c_custkey	c_custkey
nation	-	n_nationkey	n_nationkey	n_nationkey	n_regionkey
partsupp	-	ps_partkey	ps_partkey	ps_partkey	ps_suppkey
region	-	r_regionkey	r_regionkey	r_regionkey	r_regionkey
lineitem	-	l_orderkey	l_partkey	l_orderkey	l_partkey
orders	-	o_orderkey	o_orderkey	o_custkey	o_orderkey
part	-	p_partkey	p_partkey	p_partkey	p_partkey
supplier	-	s_suppkey	s_suppkey	s_suppkey	s_nationkey

Table 2: Comparisons of Total Latency for ten TPC-H queries.

	RR	Heuristics-a	Heuristics-b	CostModel	Lachesis
Environment 1	1088 sec	758 sec	939 sec	1153 sec	**672** sec
Environment 2	1121 sec	1285 sec	1002 sec	1701 sec	**944** sec

other partition approaches because it chooses to co-partition the lineitems table and the orders table which is desired by this query.

6.4 PageRank

In the PageRank application, a producer workload extracts a set of Page objects from web pages. Each Page object includes a url member that specifies the page, and a vector of urls this page links to. Then in the consumer workload, each iteration involves a `join` operation that joins the set of Page objects and the set of Rank objects. Each Rank object includes a url member, and a rank member of *double* type. we set the number of iterations to five by default, and use the default damping factor 0.85. Each PageRank iteration requires to join the ranks of links (denoted as `ranks`) with the link adjacency matrix (denoted at `links`). Then the output will be used to update `ranks`. If `ranks` and `links` are co-partitioned for the join, no shuffling is required for all iterations. Otherwise, each iteration will require shuffling.

We benchmark the PageRank application in Environment 3. The producer randomly generates and pre-processes 40 million to 100 million Page objects. Each Page object has five neighbors on average. *Lachesis* chooses to pre-partition the set of Pages and the set of Ranks using the Page object's and Rank object's url member access functions extracted from the IR. We compare *Lachesis* to round-robin partitioning and a reactive approach that co-partitions `ranks` and `links` after the first iteration. The reactive approach also relies on *Lachesis* to recognize partitioning candidates.

The results are illustrated in Fig. 10. We observe that in Environment 3, *Lachesis* can achieve up to 6.5× speedup by comparing to the round-robin partitioning; and can achieve up to 1.8× speedup by comparing to the reactive approach. In addition, when we increase

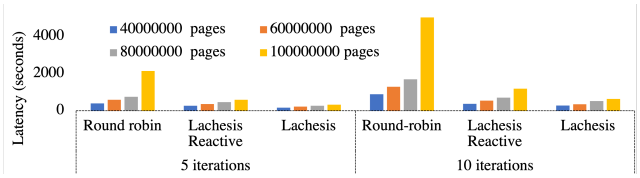
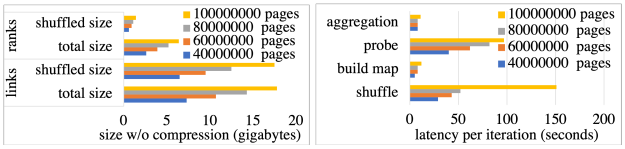


Figure 10: PageRank performance comparison



(a) Shuffled Bytes (b) Shuffling Latency

Figure 11: Shuffling in PageRank w/o Lachesis (Environment 3)

the number of iterations, the performance gain achieved by the *Lachesis*' pre-partition approach compared to the *Lachesis*' reactive approach will gradually drop, because the partitioning overhead is amortized to more iterations.

The shuffled bytes information as well as the input data size information is illustrated in Fig. 11(a). Through analysis, as illustrated in Fig. 11(b), when round-robin partitioning is utilized for processing 100 millions of pages, the shuffling of the `links` and the `ranks` accounts for **75**% of the total latency. Through pre-partitioning and re-partitioning (as in the reactive approach), the query optimizer can recognize the useful partitioning and remove these two shuffling stages. Thus all the overheads related to shuffling including hashing, data copying, and network transferring are eliminated accordingly. We also observe that the shuffling overhead increases significantly faster with the size of inputs than the rest of the overheads, because of the non-determinism in the shuffling process. For example, for shuffling the `ranks` for 100 millions Page objects, in one iteration, it takes 190 seconds on the slowest machine while less than 100 seconds on the fastest machine.

6.5 Overhead Analysis

In *Lachesis*, the overheads can be divided into three parts: the offline part that can be amortized to all partitioning requests; the online overhead for the producer that can be amortized to multiple executions of consuming workloads; and the online overhead for the consumer. In this section, we measure and analyze these overheads.

The offline overheads include creating signatures for historical IR graphs, and creating a skeleton graph from historical workflow graphs. Such overheads are sensitive to the number of workflows, number of workloads in each workflow, and number of operations in each workload. To better understand the offline overhead for large-scale workflows, we collect above statistics from real-world production workflow traces available in the publicly Workflow Trace Archive (WTA) [56]. For each trace, we generate workflow graphs following its statistics and apply our algorithms to the synthetic workflow graphs. As illustrated in Tab. 3, we find that the measured offline overhead of constructing skeleton graphs and creating signatures in one r4.2xlarge instance for the scale of real-world workflows is merely up to 14 minutes, which can be further accelerated by using multiple machines.

Table 3: Offline overhead for real-world traces. The trace names are given by WTA. WF represents the number of workflows; T represents the number of tasks in a workflow; SG-latency denotes the latency for constructing the skeleton-graph; and SN-latency denotes the latency of creating IR signatures. (latency unit: seconds)

TraceName	WF	T	SG-latency	SN-latency
S1. Askalon Old	4,583	167,677	1	1
S2. Askalon New	1,835	91,599	1	1
S3. LANL	1,988,397	475,555,927	26	12
S4. Pegasus	56	10,573	1	1
S5. Shell	3,403	10,208	1	1
S6. SPEC	400	28,506	1	1
S7. Two Sigma	41,607,237	50,518,481	717	3
S8. WorkflowHub	10	14,275	1	1
S9. Alibaba	4,210,365	1,356,691,136	94	39
S10. Google	494,179	17,810,002	8	1

Table 4: Producer Latency Comparison (unit: seconds).

data to store	w/ partition	w/o partition	overhead
15 millions of author objects	42	42	0%
78 millions of author objects	203	185	10%
20 millions of comment objects	744	726	2%
112 millions of comment objects	4, 505	4, 119	9%

At runtime, a data storage request will trigger online overheads at the producer's side that cover: (1) communicating with the DRL server, which is several milliseconds' overhead as measured; (2) dispatching the data to the storage using the partitioner automatically selected by the DRL model. This incurs up to 10% overhead as illustrated in Tab. 4, which is significantly cheaper than the shuffle operations at the consumers' side.

The online overhead at the consumer's side for processing a query involves matching of the query's IR to the partitioners associated with the input datasets to decide whether to avoid the shuffling stage. We measure this overhead by comparing the latency of enabling *Lachesis*, and simply disabling *Lachesis*. The overhead is smaller than one second for most of the workloads.

We see that compared to the significant performance speedup achieved for the consuming workloads, both of the offline and online overheads are relatively small. Particularly the offline overhead can be amortized to many runs and the online overhead is negligible. The net performance gain will be further enlarged according to the *write-once-read-many* assumption as mentioned in Sec. 3.1.

6.6 Training Overhead and Effectiveness

We choose TPC-H queries (rewritten in UDFs) [11] to create the statistics for simulating the training process. That's because it involves relatively more partitioner candidates than other workloads we have, and though TPC-H's UDFs are simple, we find the complexity of UDFs is a relatively less important feature for selecting the optimal partitionings, compared to other features (Sec 4.1.3). We first run three queries in TPC-H workload: Q01, Q02, Q04 using all possible partitioning scheme combinations, to generate statistics for actual runs, which takes 51 hours in Environment 1 with SF-10 datasets. There are in total ten partitioner candidates related to those queries, which can be used to enumerate 432 partition scheme combinations across all TPC-H datasets. For each partition scheme combination, we run the three training queries respectively in Environment 1, so that we can obtain statistics for 1296 actual runs. We also generate statistics for three different queries: Q04, Q12, Q17 in Environment 3, also using the SF-10 datasets. Because these three queries involve only a few tables, it only enumerates 17 different partitioning scheme combinations for the orders, lineitems, and parts datasets. It takes about three hours to create 51 actual runs. Using the training data augmentation technique proposed in Sec. 5, unlimited workloads can be created from these actual runs for training.

Figure 12: Training loss in Lachesis

Both the actor and critic neural networks have three fully connected layers. The first hidden layer has 128 neurons and the second hidden layer has 64 neurons. In both networks, the first two layers use leaky relu as activation function. For the output layer, the actor network uses softmax, and the critic network uses linear activation. We carefully tune the learning rate (α), entropy weight (β) and the number of neurons at the hidden layer. Fig. 12 illustrates how the training loss changes with epochs. We use a batch size of 16, and an epoch has 96 iterations. The RL-based approach takes about ten hours to run 5000 epochs with an RL server located on an r4.2xlarge instance. We find that the RL-based approach can be effective in different environments, with different data sizes, and it also requires significant manpower in training, and tuning hyper parameters like entropy value, batch size, model architecture, learning rate, etc..

For each data creation with at least two extracted partitioner candidates, we select the top three partitioner candidates (including round-robin) to formulate the state vector and action space. We find that the RL-based approach can be effective in different environments, with different data sizes, and it also requires significant manpower in training and tuning hyper parameters.

7 CONCLUSION

In this paper, we argue that automatically creating data partitionings for Big Data applications is an important and challenging task for UDF-centric workloads. We propose *Lachesis* to address the problem, which includes a unique set of functionalities for extracting, reusing, and matching of sub-computations in UDFs. *Lachesis* also provides a data placement optimizer based on a deep reinforcement learning approach and historical workflow analysis. The evaluation results demonstrate that *Lachesis* brings significant performance speedup for various UDF-centric analytics applications such as data integration, deep learning model serving, web analytics, and analytics queries. The proposed approach is effective with different data sizes and environments. Most importantly, *Lachesis* significantly reduces the efforts required on the part of enterprise IT professionals and data scientists who may not have sufficient systems tuning skills for creating partitionings for UDF-centric analytics applications.

ACKNOWLEDGMENTS

We sincerely appreciate all comments from anonymous reviewers. The work presented in this paper has been supported by the ASU FSE start-up funding, AWS Cloud Credits for Research program, Google GCP research credits program, and DARPA MUSE award No. FA8750-14-2-0270.

REFERENCES

[1] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. 2004. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 359–370.

[2] Alexander Alexandrov and et al. 2014. The Stratosphere platform for big data analytics. *VLDB* 23, 6 (2014), 939–964.

[3] Alexander Alexandrov, Andreas Kunft, Asterios Katsifodimos, Felix Schüler, Lauritz Thamsen, Odej Kao, Tobias Herb, and Volker Markl. 2015. Implicit parallelism through deep language embedding. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 47–61.

[4] Michael Armbrust, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or, Josh Rosen, Ion Stoica, Patrick Wendell, Reynold Xin, and Matei Zaharia. 2015. Scaling spark in the real world: performance and usability. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1840–1843.

[5] Wolfgang W Bein, Jerzy Kamburowski, and Matthias FM Stallmann. 1992. Optimal reduction of two-terminal directed acyclic graphs. *SIAM J. Comput.* 21, 6 (1992), 1112–1129.

[6] Matthias Boehm, Michael W Dusenberry, Deron Eriksson, Alexandre V Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R Reiss, Prithviraj Sen, Arvind C Surve, et al. 2016. Systemml: Declarative machine learning on spark. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1425–1436.

[7] Dhruba Borthakur. 2008. HDFS architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf* (2008).

[8] Yanpei Chen and et al. 2012. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *VLDB* 5, 12 (2012), 1802–1813.

[9] Zhimin Chen, Yue Wang, Vivek Narasayya, and Surajit Chaudhuri. 2019. Customizable and scalable fuzzy join for big data. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2106–2117.

[10] Stephen A Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*. 151–158.

[11] Transaction Processing Performance Council. 2008. TPC-H benchmark specification. *Published at http://www. tcp. org/hspec. html* 21 (2008), 592–603.

[12] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Cetintemel, and Stanley B Zdonik. 2015. Tupleware:" Big" Data, Big Analytics, Small Clusters.. In *CIDR*.

[13] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. 2010. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 515–529.

[14] Richard J Duffin. 1965. Topology of series-parallel networks. *J. Math. Anal. Appl.* 10, 2 (1965), 303–318.

[15] Jennie Duggan, Olga Papaemmanouil, Leilani Battle, and Michael Stonebraker. 2015. Skew-aware join optimization for array databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 123–135.

[16] Jennie Duggan and Michael Stonebraker. 2014. Incremental elasticity for array databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 409–420.

[17] George Eadon, Eugene Inseok Chong, Shrikanth Shankar, Ananth Raghavan, Jagannathan Srinivasan, and Souripriya Das. 2008. Supporting table partitioning by reference in oracle. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 1111–1122.

[18] Mohamed Y Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. 2011. CoHadoop: flexible data placement and its exploitation in Hadoop. *Proceedings of the VLDB Endowment* 4, 9 (2011), 575–585.

[19] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. 2016. Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*. PMLR, 2829–2838.

[20] K. Gupta and et al. 2011. GPFS-SNC: An enterprise storage framework for virtual-machine clouds. *IBM Journal of Research and Development* 55, 6 (2011), 2–1.

[21] HBase [n.d.]. HBase. https://hbase.apache.org/.

[22] Benjamin Hilprecht, Carsten Binnig, and Uwe Roehm. 2019. Learning a Partitioning Advisor with Deep Reinforcement Learning. *arXiv preprint arXiv:1904.01279* (2019).

[23] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a Partitioning Advisor for Cloud Databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 143–157.

[24] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. 2007. Database Cracking.. In *CIDR*, Vol. 7. 68–78.

[25] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J Gao. 2019. Declarative Recursive Computation on an RDBMS. *Proceedings of the VLDB Endowment* 12, 7 (2019).

[26] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation reuse in analytics job service at Microsoft. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 191–203.

[27] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters.. In *OSDI*. 117–134.

[28] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building efficient query engines in a high-level language. *Proceedings of the VLDB Endowment* 7, 10 (2014), 853–864.

[29] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. An intermediate representation for optimizing machine learning pipelines. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1553–1567.

[30] Joseph Lee Rodgers and W Alan Nicewander. 1988. Thirteen ways to look at the correlation coefficient. *The American Statistician* 42, 1 (1988), 59–66.

[31] Rundong Li, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Near-Optimal Distributed Band-Joins through Recursive Partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2375–2390.

[32] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[33] Harold C Lim, Shivnath Babu, and Jeffrey S Chase. 2010. Automated control for elastic storage. In *Proceedings of the 7th international conference on Autonomic computing*. 1–10.

[34] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. 2017. AdaptDB: adaptive partitioning for distributed joins. *Proceedings of the VLDB Endowment* 10, 5 (2017), 589–600.

[35] Hien Luu. 2018. Spark SQL (Foundations). In *Beginning Apache Spark 2*. Springer, 87–145.

[36] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 197–210.

[37] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*. 1928–1937.

[38] Rimma Nehme and Nicolas Bruno. 2011. Automated partitioning design in parallel database systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 1137–1148.

[39] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web*. Technical Report. Stanford InfoLab.

[40] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*.

[41] Karthik Ramachandra, Kwanghyun Park, K Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of imperative programs in a relational database. *Proceedings of the VLDB Endowment* 11, 4 (2017), 432–444.

[42] Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database management systems*. McGraw-Hill.

[43] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. 2002. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 558–569.

[44] John Riordan and Claude E Shannon. 1942. The number of two-terminal series-parallel networks. *Journal of Mathematics and Physics* 21, 1-4 (1942), 83–93.

[45] Jose F Rodrigues Jr, Agma JM Traina, Christos Faloutsos, and Caetano Traina Jr. 2006. SuperGraph visualization. In *Eighth IEEE International Symposium on Multimedia (ISM'06)*. IEEE, 227–234.

[46] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[47] Amir Shaikhha, Yannis Klonatos, and Christoph Koch. 2018. Building efficient query engines in a high-level language. *ACM Transactions on Database Systems (TODS)* 43, 1 (2018), 4.

[48] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. 2015. Clash of the titans: MapReduce vs. Spark for large scale data analytics. *Proceedings of the VLDB Endowment* 8, 13 (2015), 2110–2121.

[49] Juwei Shi, Jia Zou, Jiaheng Lu, Zhao Cao, Shiqiang Li, and Chen Wang. 2014. MRTuner: a toolkit to enable holistic optimization for mapreduce jobs. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1319–1330.

[50] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.

[51] Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*. MIT press Cambridge.

[52] tensorflow [n.d.]. FFN example with tensorflow. https://www.kaggle.com/hbaderts/simple-feed-forward-neural-network-with-tensorflow.

[53] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.

[54] Vernon Turner, John F Gantz, David Reinsel, and Stephen Minton. 2014. The digital universe of opportunities: Rich data and the increasing value of the internet of things. *IDC Analyze the Future* (2014), 5.

[55] Leslie G Valiant. 1979. The complexity of enumeration and reliability problems. *SIAM J. Comput.* 8, 3 (1979), 410–421.

[56] Laurens Versluis, Roland Math, Sacheendra Talluri, Tim Hegeman, Radu Prodan, Ewa Deelman, and Alexandru Iosup. 2020. The Workflow Trace Archive: Open-Access Data from Public and Private Computing Infrastructures. *IEEE Transactions on Parallel and Distributed Systems* (2020).

[57] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 449–462.

[58] Lanjun Wang, Shuo Zhang, Juwei Shi, Limei Jiao, Oktie Hassanzadeh, Jia Zou, and Chen Wangz. 2015. Schema management for document stores. *Proceedings of the VLDB Endowment* 8, 9 (2015), 922–933.

[59] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.

[60] T. White. 2012. *Hadoop: The Definitive Guide.* O'Reilly Media.

[61] Yahoo [n.d.]. Yahoo! Cloud Trace. https://webscope.sandbox.yahoo.com/catalog.php?datatype=s.

[62] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. 2020. Tensor Relational Algebra for Machine Learning System Design. *arXiv preprint arXiv:2009.00524* (2020).

[63] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *USENIX HotCloud*. 1–10.

[64] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. 2015. Locality-aware partitioning in parallel database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 17–30.

[65] Jingren Zhou, Nicolas Bruno, and Wei Lin. 2012. Advanced partitioning techniques for massively distributed computation. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 13–24.

[66] Jingren Zhou, Per-Ake Larson, and Ronnie Chaiken. 2010. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 1060–1071.

[67] Jia Zou, R Matthew Barnett, Tania Lorido-Botran, Shangyu Luo, Carlos Monroy, Sourav Sikdar, Kia Teymourian, Binhang Yuan, and Chris Jermaine. 2018. PlinyCompute: A platform for high-performance, distributed, data-intensive tool development. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1189–1204.

[68] Jia Zou, Arun Iyengar, and Chris Jermaine. 2019. Pangea: monolithic distributed storage for data analytics. *Proceedings of the VLDB Endowment* 12, 6 (2019), 681–694.

[69] Jia Zou, Arun Iyengar, and Chris Jermaine. 2020. Architecture of a distributed storage that combines file system, memory and computation in a single layer. *The VLDB Journal* (2020), 1–25.

[70] Jia Zou, Ming Zhao, Juwei Shi, and Chen Wang. [n.d.]. WATSON: A Workflow-based Data Storage Optimizer for Analytics. In *Proceedings of the 36th International Conference on Massive Storage Systems and Technology (MSST 2020)*.