Lachesis: Automated Generation of Persistent Partitionings for Big Data Applications

Jia Zou, Pratik Barhate, Amitabh Das, Arun Iyengar, Binhang Yuan, Dimitrije Jankov, Chris Jermaine
Arizona State University, IBM T.J. Watson Research Center, Rice University
(jia.zou, pbarhate, adas59)@asu.edu, aruni@ibm.com, (by8, dj16, cmj4)@rice.edu

ABSTRACT

Persistent partitioning is effective in improving the performance by avoiding the expensive shuffling operation, while incurring relatively small overhead. However it remains a significant challenge to automate this process for UDF-centric analytics workloads, which is closely integrated with a high-level programming language such as Python, Scala, Java. That is because user defined functions (UDFs) in such languages can contain arbitrary code that is opaque to the system and makes it hard to extract and reuse sub-computations for optimizing data placement. In addition, it is also challenging to predict the future consuming workloads. We propose the Lachesis system, which allows UDFs to be decomposed into analyzable and reusable sub-computations and relies on a deep reinforcement learning model that infers which sub-computations should be used to partition the underlying data. This analysis is then used to automatically optimize the storage of the data across applications.

1. INTRODUCTION

Big Data analytics systems such as Spark [52], Hadoop [49], Flink [3], TupleWare [12] have been designed and developed to address analytics on unstructured data, which cannot be efficiently represented in relational schemas. Users can easily represent unstructured data as nested objects, and by supplying user-defined functions (UDFs) written in the host language, such as Python, Java, Scala, or C++, users can use control structures such as conditional statements and loops to express complex computations. Such systems provide high flexibility and productivity and make it easy to develop complex analytics on top of unstructured data, which accounts for about 85% of the data on Earth [45].

For Big Data analytics workloads, data partitioning and co-location are well-known techniques to accelerate the performance of join operations [37, 2, 32, 56, 22, 16, 7, 14]. In this paper, we identify and address the problems associated with automatically creating persistent partitionings at storage time for various applications.

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxx

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
ubiquitous write-once and read-many pattern observed in production Big Data environments \cite{9, 51}, indicates that proper persistent partitioning may benefit multiple future workloads. Therefore, in this paper we argue that it is important to have automatic and persistent partitionings at storage time for UDF-centric applications: (1) when a dataset is going to be stored into the storage, the system is able to automatically determine its optimal partitionings and partition it at storage time; (2) when an application joins the dataset with other datasets, the system is able to recognize and utilize the existing partitionings to avoid unnecessary shuffling of data.

This is in nature a challenging task. First, UDFs composed in object-oriented languages, is opaque to the system. It is hard to reason with UDFs and automatically identify the desired partitioning for an application. For example, Listing\textsuperscript{2} shows a partitioner candidate that should be automatically extracted from Listing\textsuperscript{1}. This is very different with relational systems where SQL queries are easy to reason about, and a partitioning predicate is easily to be extracted (i.e. searching in "WHERE" clause) and reused (i.e. appending a "PARTITION BY" predicate).

Second, when a dataset is stored, it is often unknown which workloads will process it in the future. This is different with relational physical database design problem, which is based on a known workload of queries. Third, in popular Big Data analytics frameworks like Spark \cite{53, 52}, even if two datasets are co-partitioned at the storage layer, such information is hided from Spark, and Spark may unnecessarily re-partition the co-located datasets.

**Listing 1: UDF-Centric Join Filtering Predicate**

```java
bool join_filter (string review_line, string author_line) {
    string j = my_json::parse(review_line);
    string l_author = j["author"]->vector<r>
    r = my_csv::parse(author_line);
    string r_author = r[1];
    return (j.author == r_author);
}
```

**Listing 2: Expected partitioner for review data**

```java
string getPartitionKey1 (string review_line) {
    return my_json::parse(review_line)["author"];
}
```

In this paper, we propose a new system to automatically create persistent partitionings, called as Lachesis\textsuperscript{1}, to address the above challenges. Lachesis is built on top of the Pangea storage \cite{60, 59} with a redefined interface, in which each physical dataset is associated with the partitioner applied. Lachesis develops a lambda calculus domain specific language (DSL) that not only generates an intermediate representation (IR) for the system to understand programmer intentions, but also identifies and isolates storage-relevant sub-computations from the opaque code so that these sub-computations can be reused. Listing\textsuperscript{3} shows a UDF coded in the lambda calculus that is equivalent to Listing\textsuperscript{1} func, \texttt{==} are some of the lambda calculus constructs. The lambda calculus expression will return a tree of lambda terms as IR, as illustrated in Fig.\textsuperscript{1} from which, partitioner candidates can be automatically extracted as lambda trees. Most importantly, these partitioner candidates can be reused in the future to partition new datasets with no need for re-compilation.

**Listing 3: Listing\textsuperscript{1} expressed in lambda calculus DSL**

```java
lambda<bool> join_filter(string arg1, string arg2) {
    return func(arg1, my_json::parse("author") == func(arg2, my_csv::parse)[1]);
}
```

1 Lachesis is the name of a Greek god, who allocates lots.

The enumeration and selection of partitioner candidates for unknown future workloads is based on the recurrent workflow pattern observed in production cloud platforms. For example, 60% of the workflows in Microsoft Scope are re-executions \cite{21, 20}. As illustrated in Fig.\textsuperscript{2} when a data is going to be written to storage, Lachesis will recommend a candidate set of lambda trees based on historical workflow patterns. Then it evaluates all candidates and selects one using a deep reinforcement learning approach \cite{43}. The selected partitioner is applied to data when it is being stored. Then, when a join is about to be performed on the data, the optimizer first matches the application IR with input datasets’ partitioner, and then decides whether such partitioning can be utilized for physical optimization, e.g. to perform a local join without shuffling data.

**Our contributions** can be summarized as:

1. As to our knowledge, we are the first to systematically explore automated persistent partitioning for UDF-centric applications. We propose Lachesis, which is an end-to-end cross-layer system that automatically creates persistent partitionings at storage time to improve workflow performance.
2. We formalize our lambda calculus DSL/IR, and design and implement a set of functionalities to extract, index and persist partitioner candidates for reuse, which is a critical enabler of automatic persistent partitioning.
3. We combine the Lachesis system with a deep reinforcement learning (DRL) approach based on actor-critic networks \cite{31, 40, 30} and a historical workflow management component for choosing the optimal partitioning candidate.
4. We conduct detailed performance evaluation and overhead analysis. The results show that Lachesis can automatically generate partitionings that achieve up to 14× performance speedup for various data engineering tasks, with relatively small overhead.

## 2. SYSTEM OVERVIEW

In this section, we will first propose three important assumptions that the whole work is based on. Then we will analyze and formalize the problem. Following that, an end-to-end high-level algorithm will be given to describe our main approach. At the end, the system architecture will be presented.

### 2.1 Assumptions

The proposed Lachesis solution is hinged on three assumptions.

1. **The write-once read-many assumption** that once a data is written, it will be read many times. It indicates that creating persistent partitioning when writing the data can benefit multiple executions of workflows that take the data as input. Such pattern is observed in a number of real-world traces \cite{9}. For example, according to the publicly available Yahoo! cloud trace \cite{51}, only 17% of total stored bytes will not be accessed anymore; 60% of total stored bytes will be accessed for more than 20 times; 50% of total stored bytes will be accessed for more than 40 times, and 28% of total stored bytes will be accessed for more than 100 times.

2. **The recurrent workflow assumption** that a majority of workflows are re-executions on different or incremental datasets. Based on this assumption, we can extract partitioner candidates from historical execution of workflows, and reuse these for future datasets. For example, if a historical application \( w_i \) loads a review dataset collected in 2019 to storage; and then an application \( w_j \) joins the review dataset with the authors dataset to formulate feature vectors for product recommendation. Then in the future, if the application equivalent to \( w_j \) loads a new review dataset collected in 2020 to storage, the system may think that the workflow \( w_i \rightarrow w_j \) will repeatedly run, so \( w_j \) may process the 2020 review dataset in the future. Accordingly, the system may reuse the partitioner candidate
extracted from the application \(w_j\) to partition the new 2020 review dataset when it is being loaded to storage by the application \(w_i\). Such recurrent workflow pattern is also widely observed in recent production traces \cite{11, 20, 29}.

(3) The shared-nothing architecture assumption that we focus on the automated persistent partitioning and co-location problem in a shared-nothing distributed architecture \cite{41}, which is followed by most of high-performance, scalable, DBMSs, including Teradata \cite{50}, Netezza \cite{17}, Greenplum \cite{42}, and also used by most of the high-end internet platforms, such as Amazon, Akamai, Yahoo, Google, and Facebook \cite{56}.

\subsection{2.2 Problem Formulation}

We first present a high-level definition of the problem, as following. A producing workload \(p\) is going to write dataset \(D\), which is a set of \(n\) objects \(D = \{d_i\}, (0 \leq i < n)\) to a distributed storage system \(C\) that consists of a set of \(m\) nodes, \(C = \{c_j\}, (0 \leq j < m)\). The problem is to first find a mapping \(g : D \rightarrow C\), which is a horizontal partitioning, so that the total latency of the producer \(p\) and each of consuming workloads \(W = \{w_k\}, (0 \leq k < l)\) that take \(D\) as an input dataset (or one of the input datasets) is minimized, as denoted in Eq. (1). \(t_p\) represents the execution frequency of the consuming workload \(w_k\) in the future and \(t_i\) denote the latency of the \(i\)-th execution of \(w_k\). Then the selected mapping function \(g\) needs to be automatically applied while storing the dataset \(D\) to the cluster \(C\) for the first time.

\[
\min_{g:D \rightarrow C} \left( t_p + \sum_{\forall w_k \in W} f_{req} \sum_{i=0}^{t_i} \right) \tag{1}
\]

We can further formulate a more detailed model by lowering down the mapping functions. There exists \(m^n\) different mapping functions \(g\), to prune which, we only consider well-known horizontal partition strategies such as hash partitioning, range partitioning, round-robin partitioning, and random partitioning \cite{57}. A hash partitioner is defined by a function \(f_{hash}\) to hash \(d_i\) to a hash code \(m, \forall d_i \in D\). Range partitioning is defined similarly, except that the output key from \(f_{hash}\) must have a comparator defined for sorting; and \(g\) is accordingly defined as \(g_{range}\). Range partitioners do not require to supply any functions. The former is defined as \(g_{hash}(d_i) = \text{hash}(f_{hash}(d_i))\%m, \forall d_i \in D\). Then the selected mapping function \(g\) is also widely observed in recent studies that we focus on partitioner candidates for future datasets. The problem is to first find a mapping \(g\) to the input datasets \(D\), which is a horizontal partition strategies such as hash partitioning, range partitioning, round-robin partitioning, and random partitioning \cite{57}. A hash partitioner is defined by a function \(f_{hash}\) that extracts the partition key from each object \(d_i \in D\), where the key must have a hash function defined. For this type of partitioner, given a \(f_{hash}\) function, the corresponding mapping \(g\) is defined as \(g_{hash}(d_i) = \text{hash}(f_{hash}(d_i))\%m, \forall d_i \in D\). Range partitioners are defined similarly, except that the output key from \(f_{hash}\) must have a comparator defined for sorting; and \(g\) is accordingly defined as \(g_{range}\). Range partitioners do not require to supply any functions. The former is defined as \(g_{range}(d_i) = \text{range}(f_{hash}(d_i))\%m, \forall d_i \in D\). Round robin and random partitionings are defined similarly, except that the output key from \(f_{hash}\) must have a comparator defined for sorting; and \(g\) is accordingly defined as \(g_{random}\). Round partitioners do not require to supply any functions. The former is defined as \(g_{random}(d_i) = \text{random}(\%m, \forall d_i \in D\).

Therefore, given a set of \(q\) partition key extraction functions (i.e. partitioner candidates) on the dataset \(D\), called as \(S = \{f_q\}, 0 \leq q < q\), the search space is re-defined as \(S^2 = \{g_{hash}\} \cup \{g_{range}\} \cup \{g_{random}\}\), \(\forall f_q \in S\). Then Eq. (1) can be lowered into Eq. (2).

\[
\min_{g:S^2} \left( t_p + \sum_{\forall w_k \in W} f_{req} \sum_{i=0}^{t_i} \right) \tag{2}
\]

Different with relational physical database design problem \cite{37, 2, 32, 56, 23}, we are not given the set of workloads \(W\), which complicates following problems:

(1) How to obtain the set of partitioner candidates \(F\), with each partitioner candidate executable and applicable to the dataset \(D\) at storage time?

(2) How to solve the optimization problem illustrated in Eq. (2)?

\subsection{2.3 Problem Analysis and Main Approach}

\textbf{Workload (W) enumeration.} We introduce a set of historical workloads, denoted as \(W = \{w_i\}, (0 \leq i < \text{nw}\). Each workload record \(w_i = (\text{binary}, \text{input} = \{\text{path}\_\text{input}_i\}, \text{output} = \{\text{path}\_\text{output}_i\}) \in W, (0 \leq i < \text{nw}, 0 \leq k < \text{ni}, 0 \leq l < \text{no})\) consists of the binary executable code (\text{binary}), a list of the path information for the input datasets (\text{input}), and for the output dataset (\text{output}). Then based on the recurrent workflow assumption, we can conclude that: Given a producer workload \(p\), if \(\exists w_i \in W, \text{satisfying} p.\text{binary} = w_i.\text{binary}\), and if \(\exists w_i \in W, k.i, \text{satisfying} w_i.\text{path}\_\text{output}_i = w_i.\text{path}\_\text{input}_i\), we have \(w_i \in W\).

\textbf{Partitioner Candidate (F) enumeration.} Suppose there exists a mapping function \(h : W \rightarrow A\) provided by a DSL/IR subsystem that translates each opaque workload \(w_i.\text{binary}, w_i \in W\) into a directed acyclic graph \(a_i = (V; E; S; O) \in A\), so that \(h(a_i) = a_i\). Each node \((v \in V)\) represents an executable atomic computation (i.e. lambda term), with a tag \text{partition}\_\text{required} to specify whether the computation requires a partitioning operation. Each edge \((e \in E)\) represents a dataflow from the source node to the destination node, or a control flow so that the execution of the destination node depends on the output of the source node. \(S \subseteq V\) is the set of all input scanner nodes directly reading from input datasets. \(O \subseteq V\) is a set of output nodes that produce the final outputs from the workload.

Then \(\forall w_i \in W\), given a dataset \(D \in w_i.\text{input}\), the DSL/IR guarantees that we can easily locate its corresponding scanner node \(s_i \in S\) by simply checking each scanner node’s input data path. (We encapsulate this process as a simple function called \text{findScanner}(\cdot))

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
 & RDBMS & CoHadoop & Spark & SystemML & Lachesis \\
\hline
Persistent partitioning & ✓ & ✓ & ✓ & ✓ & ✓ \\
Automatic partitioning & ✓ & ✓ & ✓ & ✓ & ✓ \\
UDF-centric programming & ✓ & ✓ & ✓ & ✓ & ✓ \\
\hline
\end{tabular}
\caption{Comparison of partitioning capabilities.}
\end{table}
Then we can enumerate its partitioner candidates as a set of subgraphs of $a_i$, denoted as $F_i$. Each $f_k = \{V_k, E_k, S_k, O_k\} \in F_i$ defines a subgraph of $a_i$, satisfying $V_k \subseteq V, E_k \subseteq E, S_k = \{s_j\}$, and $|O_k| \leq 1$. The constraints over $S_k$ and $O_k$ indicate that the subgraph is either empty, or has only one scanner node ($s \in S_k$) corresponding to $D$ and only one output node ($o \in O_k$) that outputs the partition key for a partition-required node.

Such subgraphs can be efficiently identified through two steps. The first step is to recursively traverse $a_i$ and enumerate all distinct paths that are started at $s_j$ and ended at one of the partition-required nodes, if any, as illustrated in Alg. 1. The second step is to merge all paths that connect the same scanner node and the same partition-required node into one graph to serve as one partitioner candidate, as illustrated in Alg. 2.

If we have a DSL/IR subsystem that serves as $h : W \rightarrow A$, we can obtain a set of graph IRs denoted as $A$. More importantly, $\forall a_i \in A$, we can further retrieve a set of partitioner candidates (as subgraphs of $a_i$). Thus we can formulate a function $h_{W \rightarrow F}$ as illustrated in Alg. 2.

**Algorithm 1** $\text{search}(a_i, s_j, F_i)$

1: INPUT: $a_i = \{V, E, S, O\} \in A$; $s_j; F_i = \emptyset$
2: OUTPUT: a list of partial partitioner candidates for input $s_j$ extracted from $a_i$
3: $V' \leftarrow s_j; E' \leftarrow \emptyset; S' \leftarrow \{s_j\}; O' \leftarrow \emptyset$
4: for $v_k \in s_j$. children do
5: if ($v_k$.partition_required) then
6: $V' \leftarrow V' \cup \{v_k\}$
7: $E' \leftarrow E' \cup \{edge(s_j, v_k)\}$
8: $O' \leftarrow O' \cup \{\}$
9: $F' \leftarrow \emptyset$
10: $\text{search}((V - V', E - E', S - S', \{v_k\}, O', F'))$
11: end for
12: end if
13: $F_i \leftarrow F_i \cup \{(V', E', S', O')\}$
14: end for
15: $\text{return } F_i$

**Algorithm 2** $\text{merge}(F_i)$

1: INPUT: $F_i$ (a list of partial partitioner candidates output from Alg. 1)
2: OUTPUT: a list of partitioner candidates
3: $\text{hashmap} \leftarrow \emptyset$
4: $P_i' \leftarrow \emptyset$
5: for $p_k = \{V_k, E_k, S_k, O_k\} \in F_i$ do
6: if $\text{hashmap.count}(\{S_k, O_k\}) \neq 0$ then
7: $\{V', E', S', O'\} \leftarrow \text{hashmap}(\{S_k, O_k\})$
8: $f_{\text{updated}} = (V' \cup V_k, E' \cup E_k, S_k, O_k)$
9: $\text{hashmap}(\{S_k, O_k\}) \leftarrow f_{\text{updated}}$
10: else
11: $\text{hashmap}(\{S_k, O_k\}) \leftarrow f_k$
12: end if
13: end for
14: for $(S_k, O_k), f_k \in \text{hashmap}$ do
15: $F_i' = F_i' \cup \{f_k\}$
16: end for
17: $\text{return } F_i'$

A DRL-based Optimization Approach to Eq. 2 The selection of the optimal partitioner candidate to be applied at storage time depends on a number of hard-to-predict factors, such as the frequency and latency of each consuming workload that contains a certain partitioner candidate, as well as the value and size distribution of the partition key output from the partitioner candidate, and the partitioner candidate’s computational complexity. Data schema evolution, hardware updates, and changes in workload characteristics may also affect above factors. Therefore, instead of using a cost-based searching approach, we choose to use deep reinforcement learning to solve the optimization problem defined in Eq. 2. We give an end-to-end algorithm (Alg. 3) that formalizes the high-level process for creating a persistent partitioning at storage time as illustrated in Fig. 2.

**Algorithm 3** $\text{persistentPartition}(p_i, D, W')$

Require: the producer workload: $p_i$; the dataset to write: $D$; the set of historical workloads: $W'$.
1: $W \leftarrow \text{match}(p_i, W')$ (Sec. 4)
2: $T \leftarrow T$
3: for $w_k \in W$ do
4: $a_i \leftarrow h(w_i)$ (Sec. 3)
5: $s_j \leftarrow \text{findScanner}(D)$
6: $F_i \leftarrow \text{merge}((a_i, s_j, \emptyset))$ (Alg. 1 and Alg. 2)
7: $T \leftarrow T \cup F_i$
8: end for
9: $g \leftarrow \min_{d \in T} \{t_p + \sum_{w_k \in W} (freq_k \times t_k)\}$ (Sec. 5)
10: for $\forall d_i \in D$ do
11: $\text{store } d_i$ to the node $g(d_i)$
12: end for

2.4 System Architecture

In Lachesis, we redefine the storage and computation interfaces, so that on one hand, the computation layer can pass the partitioner to storage for creating the optimal persistent partitioning at storage time; and on the other hand, the storage can pass applied persistent partitioning and statistical information to the computation layer at runtime for query optimization.

The main components of the system include:

(1) The Lambda Calculus DSL/IR. Lachesis includes a new DSL that requires a programmer to write UDF-centric codes in a fully declarative variant of the relational algebra [38]. For example, a join operator can take any number of arbitrary datasets as inputs in the Lachesis DSL. When the programmer implements such a join, she must provide a filtering UDF (where we can easily locate partitioning candidates) and a projection UDF, both of which are written in a special lambda calculus [29] [8]. This lambda calculus is designed so that it isolates atomic operations, asking the programmer to supply lambda terms over user-defined types (such as equality checks, method invocation, member variable extraction, etc.). These individual lambda terms/atomic operations can each be compiled into efficient machine code or bytecode at the time the DSL program is compiled, and never need to be looked into again. The entire DSL program is compiled into a persistent IR, which is a workflow of executable lambda terms. When performing optimizations over a workload (such as automatically choosing co-partitioning strategies), Lachesis needs only to analyze the persistent historical IRs and extract the lambda terms as partition candidates.

(2) The Historical Workflow Analyzer. Lachesis stores various information regarding historical application executions, including the path to the input datasets, and the output datasets, location of the IR, runtime statistics such as input data size, output data size, execution latency, execution latency for each job stage, and so on. The producer and consumer relationships among applications are reconstructed and re-executions of workloads are detected and labeled, which provides a full picture of historical workflow executions. The
main idea is that given a dataset to be stored by its producer application, the historical workflow analyzer can efficiently supply a subset of historical applications that once have processed output datasets from the same producer application. Based on the recurrent workflow assumption, each application in the subset may rerun and process the dataset in the future. Therefore, any relevant partitioning computations extracted from these applications can serve as a partitioner candidate for this dataset. In further, the historical workflow analyzer will extract feature vectors for all partitioner candidates to feed to a deep reinforcement learning model as the state vector for the final selection. More details are described in Sec. 4.

(3) The Partitioner Selector, *Lachesis* trains a deep reinforcement learning model (DRL) using the actor-critic network [43] to serve as the partitioner selector. Each time a dataset is to be stored, the historical workflow analyzer recommends a set of partitioner candidates, and sends a state vector that describes various features for each of the candidates, to a TensorFlow-based DRL server. The DRL server predicts and samples a probability distribution over an action space that consists of all candidates to select a partitioner candidate, which is then applied to partition the dataset at storage time. The overall throughput variation for recent workloads are computed to serve as the reward. The DRL model is able to automatically adapt to changes in hardware environments, workload characteristics and data schemas by learning from the historical rewards. More details are described in Sec. 5.

There are other components in the *Lachesis* system, including a data rebalancer, which adjusts existing data partitionings in case of node removal, node addition, changes in workload characteristics, and evolution of data schema, by moving only a portion of data across the cluster nodes; and a data replicator, which creates heterogeneous replications [59] of data by selecting and applying multiple persistent partitionings. Because these components are not in the critical path of the automatic partitioning workflow, we omit the details in this work.

Although we focus on the persistent partitioning problems, our work can be easily extended to automate intra-application partitioning, and construction of materialized views, indexes, and other workload-wide optimization tasks.

3. **LAMBDA CALCULUS DSL/IR**

We provide a set of subgraph-reuse-friendly DSL/IR functionalities to serve as the function \( h_{\text{Wf}} \cdot A \) that extracts from an opaque workload a representation of the more detailed data flows and control flows to describe and unobscure the workload. The main idea is to embed a DSL in a high-level programming language, and the compilation of the DSL generates an IR. The design of DSL/IR targets at three goals:

1. The DSL must be fully declarative and easy to use, so that the programmer only specifies what to compute, and the system optimizer will carry out how to compute (including the selection of join ordering, selection of the partitioner), based on the hardware environments, input data characteristics, etc. This ensures the data independence of our proposed automatic partitioning process.

2. The IR must facilitate the reuse of a connected subgraph of the IR so that it can run separately like a standalone executable. This ensures that \( \forall F_i \in \mathcal{F} \), \( F_i \) can be reused independently.

3. The IR must facilitate the analysis of the desired partitioning logic for each partition-required operator, so that the implementation of the partitioner candidate enumeration function \( h_{\text{Wf}} \cdot \cdot \) is feasible.

To fulfill above goals, the corresponding design decisions made are as follows:

1. In the DSL, we provide a set of high-level declarative operators such as join, aggregate, select (i.e., map), multiselect (i.e., flatten), sort, partition and so on, which can be further customized by UDFs composed by lower-level declarative lambda calculus constructs [6]. Although it look similar to dataflow languages such as Spark [53] with Weld [34] embedded (so that our DSL is as easy to use as these languages), there is a distinction in the declarativeness. For example, the join can take any number of collections of arbitrary objects as inputs with lambda calculus specifying how to join. This is different with the join in other dataflow languages, which can only take two collections of key-value pairs, and leads to an undesirable fact that the join ordering must be controlled by the programmers.

2. To facilitate the reuse of any subcomputation embedded in UDFs, the IR is designed into a DAG where each node represents an atomic computation (i.e., a lambda term) that can be isolated and executable. Once the DSL is compiled, the binary codes for these atomic computations are generated. Therefore, it is easy to reuse and execute a subgraph that combines arbitrary atomic computations. As far as we know, we are the first to propose such IR design to facilitate the decomposition of UDFs and the reuse of their subcomputations. To do the same in existing dataflow languages, you need analyze and glue assembly code together at LLVM level, or glue high-level DSL fragments together and then compile these to executable code [12, 34].

3. To facilitate the analysis of subcomputations, we allow programmers to indirectly control the number of nodes in an IR graph to a reasonable level by encapsulating a bulk of user defined logic not relevant with partitioning (e.g., a complex UDF that specifies join projection logic) into an opaque lambda term.

3.1 **Formalization**

Our DSL is mainly based on two core data abstractions: typed Set and typed Map. All data involved in a Big Data analytics workflow can be represented using these two data structures. A Set represents a collection of immutable objects that share the same type, such as the input data, and output data. It can be associated with a partition scheme (i.e., range partition, hash partition, random partition, round-robin partition, customized partition) and a key projection function that extracts the partition key from each object, which is required for range partition and hash partition.

\[
Set(T, \text{partitionKeyT}) \equiv \\
\begin{cases} 
\text{isPartitioned: } \text{bool}; \\
\text{partitionScheme: } \text{enum}\{\text{Hash, Range, } \ldots\}; \\
\bar{f}_{\text{keyProj}} : T \rightarrow \text{partitionKeyT}; \\
D : \text{Vector}(T); 
\end{cases}
\]

In addition, a Map is used to represent a hash map of (key, value) pairs generated immediately for aggregation and join probing, which can be mutated, described as following. The (key, value) pairs are always hash partitioned on its key.

\[
\begin{cases} 
\text{Map}(\text{KeyT}, \text{ValueT}) \equiv \\
D : \text{HashMap}(\text{KeyT}, \text{ValueT}); 
\end{cases}
\]

Some examples of the formalized relational operators are described as following:

**Partition-Required Operators**

(1) A join operator that takes arbitrary number of inputs, a filter function \( f_{\text{fil}} \) (i.e., the WHERE clause) and a transformation function \( f_{\text{proj}} \) (i.e., the SELECT clause), maps \( n \) input Sets of objects having type \( T_1, \ldots, T_n \) to an output Set of objects having type \( O \):

\[
\begin{align*}
\text{join}(T_1, \ldots, T_n, O) & \equiv \\
\begin{cases} 
\text{f}_{\text{fil}} : \{T_1, \ldots, T_n\} \rightarrow \text{bool}; \\
\text{f}_{\text{proj}} : \{T_1, \ldots, T_n\} \rightarrow O(S_1, \ldots, S_n) = \{\text{f}_{\text{proj}}(v_1, \ldots, v_n) | f_{\text{fil}}(v_1, \ldots, v_n) = \text{true}, v_i \in S_i, i \in 1..n\}
\end{cases}
\end{align*}
\]
(2) An aggregate operator takes a key projection function \( f_{\text{keyProj}} \), a value projection function \( f_{\text{valueProj}} \), an aggregation function \( f_A \) that aggregates two values that have the same key, and an output function \( f_O \) that constructs output object having type \( O \) from an aggregated \((k, v)\) pair. In the end, it transforms an input Set having type \( T \) to an output Set of objects having type \( O \):

\[
\text{aggregate}(T, K, V, O)\{ \\
f_{\text{keyProj}}: T \rightarrow K, f_{\text{valueProj}}: T \rightarrow V, \\
f_A: (V, V) \rightarrow V, \\
f_O: (K, V) \rightarrow O(x) = \{ f_O(k, f_A(\{(k = f_{\text{keyProj}}(a), v = f_{\text{valueProj}}(a), a \in x\})) \}
\]

In practice, two functions are implemented based on \( f_A \): a local aggregation function that aggregates a local \((k, v)\) pair to an intermediate Map\((K, V)\) structure, and a global aggregation function that aggregates a shuffled Map\((K, V)\) structure to a same Map\((K, V)\) structure that conducts final aggregation for a data partition.

(3) A sort operator takes a comparator function \( f_{\text{keyProj}} \) and sort the input Set of objects according to the the key (of type \( K \), on which a comparator function must be defined) projected from each object:

\[
\text{sort}(T, K, V, O)\{ \\
f_{\text{keyProj}}: T \rightarrow K \}(S) = \{v|v \in S\}
\]

(4) A partition operator takes a key projection function \( f_{\text{keyProj}} \) and partitions the input Set of objects according to the key (of type \( K \), on which a hash function must be defined) projected from each object:

\[
\text{partition}(T, K, V, O)\{ \\
f_{\text{keyProj}}: T \rightarrow K \}(S) = \{v|v \in S\}
\]

Non-Partition-Required Operators

(1) A select operator that takes a filter function \( f_{\text{fil}} \) (i.e. the WHERE clause) and a transformation function \( f_{\text{proj}} \) (i.e. the SELECT clause), maps an input Set of objects \( S \), having type \( T \) to an output Set of objects having type \( O \):

\[
\text{select}(T, O)\{ \\
f_{\text{fil}}: T \rightarrow bool, f_{\text{proj}}: T \rightarrow O \}(S) = \{f_{\text{proj}}(v)|v \in S, f_{\text{fil}}(v) == \text{true}\}
\]

(2) A multiselect operator flattens an input Set of objects having type \( T \) to an output Set of objects having type \( O \):

\[
\text{multiselect}(T, O)\{ \\
f_{\text{fil}}: T \rightarrow bool, f_{\text{proj}}: T \rightarrow T, \text{Vector}(O) \}(S) = \{\text{true}, v^\prime \in f_{\text{proj}}(v)|v^\prime \in S, f_{\text{fil}}(v) == \text{true}\}
\]

To make functions (e.g. \( f_{\text{partition}}, f_{\text{fil}}, f_{\text{compare}}, \text{and etc.} \)) decomposable to executable partitioner candidates, we further propose a set of lambda calculus expressions to define these functions.

The expressions consist of a set of built-in lambda abstraction families [29] to create lambda terms for input objects, as well as a set of higher-order functions [8] that take as input one or more lambda terms, and return a new lambda term. These built-in lambda abstraction families include:

(1) member\((x, \text{attribute})\), takes an object \( x \) as input, and returns a function returning one of the object’s member variables (i.e. \( \lambda x.(x\_\text{attribute}) \)) in lambda calculus formalization [6]. Here, the first \( x \) is the variable, and the part after . is the function to be applied to the variable.;

(2) method\((x, \text{methodName})\), which returns a function calling a method on the object (i.e. \( \lambda x.(x\_\text{methodName}(x)) \));

(3) literal\((I)\), returns a function returning \( I \) (i.e. \( \lambda x.I \));

(4) self\((x)\), returns an identity function (i.e. \( \lambda x.x \));

(5) func\((x, f)\), returning a function calling opaque user defined function \( f(x) \) to express complex but not performance-critical logic, for balance of ease of user programming and ease of system query optimization (i.e. \( \lambda x.f(x) \)).

Then a set of higher-order functions are provided to compose above basic lambda terms into a new lambda term that can be regarded as a tree of lambda terms, which include:

(1) The boolean comparison operations: \( ==, >, !=, \text{etc.}\);

(2) The boolean operations: \&\&, | |, !, \text{etc.};

(3) The arithmetic operations: +, -, *, \text{etc.};

(4) The construction operation: construct\((O, l_1, \ldots , l_n)\), which returns a lambda term that constructs a new object of type \( O \) using values returned from lambda terms \( l_1, \ldots , l_n \);

(5) Parentheses that define the explicit ordering for lambda term composing: \((())\);

(6) index operator: \[\] .

(7) conditional branch: \( l_1 ? l_2 : l_3 \), or if\((l_1) l_2 \text{ else } l_3 \).

About manual and intra-application partitioning. Besides the automatic persistent partitioning that we advocate in our work, based on our DSL, programmers can also choose to manually create persistent partitions. If a set is defined with a partition scheme identifier (enum type, which can specify one of range partitioning, hash partitioning, random partitioning, and round-robin partitioning); and optionally a partition computation (a lambda term object) \( f_{\text{keyProj}} \) that specifies how to extract the partition key from each object of type \( T \) in the set. Also the partition operator can be used to perform an intra-application partitioning.

3.2 IR Functionalities

At execution time, UDF-centric applications coded in our declarative DSL will be translated into an IR, which is a directed acyclic graph (DAG). Each node is an executable atomic computation, which is annotated with the information regarding this atomic computation (e.g. computation type, invoked method name, projected attribute name, etc.). Each edge represents a dataflow, for which the destination node processes the data output from the source node; or a control flow, where the execution of the destination node depends on the output from the source node. For example, the IR created for the code in Listing 3 is illustrated in Fig. 1.

The supported atomic computations includes unary physical operators such as to apply a lambda abstraction function to or create hash on a vector of objects; and binary physical operators such as to compare two vectors of objects, to join two vectors of hashed objects, and to filter a vector of objects based on an associated vector of boolean values. These lower-level atomic computations are fully transparent to the programmers. We implement each atomic computations using C++ meta templated programming for high-performance.

As mentioned, the IR is generated while compiling the DSL. So each subgraph can be executed separately without the need for re-compilation. For example, the two partitioner candidates highlighted in Fig. 1 can be executed directly from the IR, requiring no compilation and code generation. Thus, any partitioner candidate can be reused if the corresponding historical IR exists. Easy reuse of historical IR subgraphs is a critical enabler for automatically creating partitionings at storage time.

We propose several unique IR functionalities for creating persistent partitioning. The first functionality is to extract and index a subgraph of the IR as a partitioner candidate using the source scanner node and the output node that are unique for each partitioner candidate. The second functionality is to store and cache historical IRs and subgraph indexes, which facilitates efficient reuse of partitioner candidates. The third one is an IR matching functionality that is often used in query optimization, for determining whether the partitioner of the input datasets matches the desired partitioner of an partition-required operator, so that it can be utilized to avoid a shuffling operation. The IR matching is similar to a DAG iso-
In this section, we answer two questions:
1. Given a producer $p_i$ that is going to write a dataset $\mathcal{D}$ to the storage, and a set of historical workloads $\mathcal{W}$, how to enumerate the set of workloads $\mathcal{W}$ that may process $\mathcal{D}$ in the future?
2. Given a set of partitioner candidates $\mathcal{F}$ extracted from $\mathcal{W}$, how do we extract feature vectors for each $F_i \in \mathcal{F}$ based on historical workflow analysis?

The main idea to address the first question is based on the recurrent workflow assumption as described in Sec. 2.1. A workflow is a DAG, where each node represents an application (i.e., workload), as illustrated in Fig. 3. We first reconstruct low-level workflow information from execution logs. As illustrated in Fig. 3(a), each node represents one execution run of an application/workload, and each edge represents a historical dataset created by its source workload run, and consumed by its destination workload run.

We will further condense the low-level graph into a super graph by grouping nodes that have the same IRs and thus expect exactly the same persistent partitionings, as illustrated in Fig. 3(b). In the super graph, each edge represents a list of historical execution runs in form of (IR_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 super graph in Fig. 3(b), Lachesis will predict that applications from group2 and group1 may process the dataset in the future.

Then the system will enumerate partitioner candidates from these groups as described in Sec. 3. To select a partitioner candidate using the DRL model as described in Sec. 5, the system will extract features for each partitioner candidate based on historical workflow analysis. These features include:
1. frequency indicates the total number of runs of the IR where the partitioner candidate is extracted from.
2. distance indicates the average time interval between the most recent $k$ runs in the candidate’s group.

In this work, we investigate whether the RL approach can also work for this new problem by utilizing the features proposed in Sec. 4 to encode each partitioner candidate.

5. PARTITIONER SELECTION

In this section, we attempt to solve the optimization problem as illustrated in Eq. 2 using a reinforcement learning (RL) approach. There are existing works targeting at similar data partitioning optimization problems in OLAP relational databases [37, 2, 32, 39, 19, 18, 15, 54, 26]. These works, including existing RL-based partitioning advisors [19, 18], are largely depending on the functional dependency across attributes in relational database. Such dependency does not exist in many semi-structured data and non-structured datasets, which are ubiquitous in UDF-centric analytics. In this work, we investigate whether the RL approach can also work for this new problem by utilizing the features proposed in Sec. 4 to encode each partitioner candidate.

5.1 Reinforcement Learning (RL)

In this paper, we propose an RL-based approach to make the partitioner selection in Lachesis adaptive to environmental dynamics by learning through the rewards for past actions. Fig. 4 illustrates Lachesis’ automatic partitioning approach based on RL.

Each time an application is about to load data or materialize output, a set of candidate partitioners are obtained as described in Sec. 3 based on historical workflow analysis.

Then Lachesis generates a “state” vector that includes the features of all obtained partitioner candidates to represent the current environment. Once a partitioner candidate is chosen by the RL model, the partitioner will be applied to create the dataset, and Lachesis also computes the reward by measuring the throughput of runs of applications that consumes the partitioned dataset and compare the throughput with a baseline measured as the average of all historical runs of these applications in a time period.
It sends “reward” and “state” in a JSON message to Lachesis’s RL server, which implements a neural network in Python based on TensorFlow APIs. The RL server selects a candidate using the neural network and sends its index back.

5.2 RL Model

Lachesis uses the actor-critic network \([49]\), which is an RL approach mainly based on policy gradient, to model the adaptive data partition problem. The optimization goal of the model is to maximize cumulative processing throughput of current and future applications.

The actor network takes “state” as input, and outputs “policy”, which is represented as the probability distribution in the “action” space. In the same time, 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 that improves the learning for both of the actor network and the critic network.

The RL model is deployed in a separate Python server that relies on TensorFlow and works as an RL agent. The Python server accepts the candidate lambda features and measured performance resulted on TensorFlow and works as an RL agent. The Python server accepts the learning for both of the actor network and the critic network.

To do training based on above statistics, we randomly generate traces from actual runs of a few TPC-H queries \([10]\) and identify all the partitioner candidates that can be used for partitioning input datasets as well as their feature vectors. For each partitioner candidate \(L_i\), we can obtain statistics such as reference distance of each query \(Q_i\) (denoted as \(d_i\)), occurrences of each query \(Q_j\) (denoted as \(f_j\)), recency of each query \(Q_j\) (denoted as \(r_j\)), complexity of the partitioner candidate (denoted as \(c_i\)), average selectivity of the partitioner candidate in historical \(Q_i\) (denoted as \(s_i\)), and average number of distinct keys created by the partitioner candidate for historical \(Q_i\) (denoted as \(k_i\)).

For each partition scheme, we run the selected queries against the datasets partitioned with all candidate schemes and measure the latency for each run.

To do training based on above statistics, we randomly generate workloads by sampling any combinations from the selected TPC-H queries. For example, \(\{Q_1, 0.5\}, \{Q_2, 0.5\}\) represents a workload that consists of two queries \(Q_1\) and \(Q_2\), with same frequency.

Then for each workload, not only the overall latency for different partition schemes can be estimated from the measured latency of historical runs, the feature vector for lambda term candidates can also be derived from historical statistics. In above example, if \(Q_1\) uses a set of partitioner candidates \(\{L_1, L_2\}\), and \(Q_2\) uses a set of partitioner candidates \(\{L_2, L_3\}\), then we can generate a set of partitioner candidates for this workload as \(\{L_1, L_2, L_3\}\). For \(L_1\), its feature vector is \((d_1, f_1, r_1, c_1, s_1, k_1, ...);\) for \(L_2\), its feature vector is \((\text{avg}(d_1, d_2), \text{avg}(f_1+f_2), \text{avg}(r_1+r_2), e_2, \text{max}(s_21, s_22), \text{min}(k_{21}, k_{22}), ...));\) and for \(L_3\), its feature vector is \((d_3, f_{31}, s_{32}, k_{33}, ...);\). Here, because \(L_2\) is applied to both \(Q_1\) and \(Q_2\), the construction of its feature vector need consider statistics of \(L_2\) for both queries.

We choose to use the maximum value for selectivity and use the minimum value for number of distinct keys, mainly because we want to encourage partitioning using partitioner candidates for job stages that have large selectivity and avoid partitioning using lambda terms that leads to small number of distinct hash keys.

In this way, we can generate unlimited number of workloads, and generate large volume of training data.

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 \([30]\):

\[
\nabla_\theta E[\sum_{t \geq 0} \gamma^t r_t | s_0] = E[\nabla_\theta \log \pi_\theta(s, a) A^\theta(s, a) | s_0]
\]

\(A^\theta(s, a)\) 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:

\[
\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) A(s_t, a_t) + \beta \nabla_\theta H(s_t).
\]

Here, \(\alpha\) is the learning rate; \(H(\cdot)\) is the entropy of the policy, which is to encourage exploration in the action space; and \(\beta\) 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 \(V^\pi_\theta(s) = Q(s, a)\). 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 \([30]\).

5.3 Training Methodology

Lachesis runs a training phase in which the RL client explores a Big Data processing environment. Ideally, training would occur with actual data loading and workload execution. However, this will be slow as the RL server must wait until all of the datasets are loaded with the partition schemes specified by last actions and related queries are all executed.

To accelerate the training process, we generate (state, action, reward) traces from actual runs of a few TPC-H queries \([11]\) and identify all the partitioner candidates that can be used for partitioning input datasets as well as their feature vectors. For each partitioner candidate \(L_i\), we can obtain statistics such as reference distance of each query \(Q_i\) (denoted as \(d_i\)), occurrences of each query \(Q_j\) (denoted as \(f_j\)), recency of each query \(Q_j\) (denoted as \(r_j\)), complexity of the partitioner candidate (denoted as \(c_i\)), average selectivity of the partitioner candidate in historical \(Q_i\) (denoted as \(s_i\)), and average number of distinct keys created by the partitioner candidate for historical \(Q_i\) (denoted as \(k_i\)).
Then the training component works like a database simulator, it first generates a workload, derives the partition candidates, and forms the state vector from features of these candidates and the environment, all based on historical statistics. Then it sends the state vector to the RL server and obtains the action for partitioning data. Instead of really partitioning the data and running the workload, it directly computes reward from historical latency statistics and sends back reward to the RL server.

6. Evaluation

In this section, we mainly want to answer following questions: (1) What is the performance gain that can be achieved by Lachesis’s automatic persistent partitioning for different types of Big Data analytics applications, including data integration workflow, analytics workflow, linear algebra operations, and relational queries on different hardware platforms? (Sec. 6.2) (2) How much online and offline overhead is incurred during the automatic data partitioning process? (Sec. 6.3) (3) How effective is the training process? (Sec. 6.4) (4) How will the amount of history affect the effectiveness? (Sec. 6.5) (5) What are the issues with data partitioning in Spark? (Sec. 6.6)

6.1 Environment Setup

We implement Lachesis on top of a baseline system, which is PlinyCompute [58], a UDF-centric analytics framework, using the Pangea storage [59]. When Lachesis is enabled, all job execution information will be logged, and each data loading or data materialization computation will trigger a query to the data placement optimizer. To measure the performance gain brought by automatic persistent partitioning, we implement a set of representative workloads on the baseline system, including: (1) Reddit data integration workflow. We implement workflow that involves three workloads: the first workload loads the Reddit authors data to the storage, the second one loads the Reddit submissions data to the storage, and the third one joins the authors data with the submissions data, and outputs integrated data. (2) PageRank Analytics workflow. We implement a web search analytics workflow that involves two workloads: pre-processing the web pages, and running the page rank algorithm on the pre-processed web pages [33]. (3) Linear Algebra Operations. We mainly select four representative linear algebra workloads: dense matrix multiplication, sparse matrix multiplication, gram matrix (given a matrix \(X\), compute \(X^T X\)) [58], and the least squares linear regression (given a matrix of features \(X\) and responses \(y\), compute \(\hat{\beta} = (X^T X)^{-1} X^T y\)) [58].

The producer workload loads matrix data into the storage, which will be processed by above mentioned linear algebra operations. (4) TPC-H Queries. We mainly select ten TPC-H queries as consuming queries. The producer workloads load seven tables into the storage, and then these consuming queries will run to process the loaded data.

We mainly use three cluster environments. One six-node cluster with 10 Gbps network bandwidth, one eleven-node cluster with 10 Gbps network bandwidth, and one eleven-node cluster with lower than 1 Gbps network bandwidth, all deployed on the AWS EC2 platform. In addition, we also use one nine-node cluster with 10 Gbps network bandwidth in Google Cloud Platform (GCP). For the two AWS high-network-speed clusters, each node is a r4.2xlarge instance that has 8 CPU cores, 68GB memory, and 100 GB EBS SSD for persistent storage. For the AWS low-network-speed cluster, each node is a m2.4xlarge instance that has 8 CPU cores, 68GB memory, and 100 GB EBS SSD for persistent storage. For the GCP cloud, each node has 8 CPU cores, 52 GB memory, and 100 GB SSD storage. For all clusters, one node is used as the master and the rest of nodes are used as workers. We expect that Lachesis should be adaptive to different platforms. In Sec. 6.5 and Sec. 6.6, we also use a small scale cluster that consists of one master and two workers, which are all r4.2xlarge instances.

6.2 Performance Evaluation

In this section, we will demonstrate the performance speedup brought by Lachesis for consuming workloads that have input datasets automatically partitioned. We measure the performance speedup by comparing the consuming workload’s latency of the case where the input datasets are automatically partitioned by Lachesis at storage time, and the case where the input datasets are directly written to storage without any data placement optimization.

6.2.1 Reddit Data Integration Workflow

In this workflow, one workload is responsible for loading the Reddit submissions data in JSON format into the Pangea storage as a Set of Submission objects; one workload is responsible for loading the Reddit authors data in CSV format into the Pangea storage as a Set of Author objects; and the third workload is responsible for joining each submission object with its corresponding author objects. By using Lachesis, when the submissions and authors are loaded into the storage, they are automatically partitioned by each object’s author information. In addition, when executing the third workload, the system can recognize the persistent partitionsings applied to the input datasets and the shuffling operation is avoided.

For the experiments, we mainly test two cases. (1) Small data case: Loading and joining 20 millions of Reddit submissions with 15 millions of Reddit authors using two AWS r4.2xlarge instance nodes as workers. (2) Large data case: Loading and joining 112 millions of Reddit submissions with 78 millions of Reddit authors using ten AWS r4.2xlarge instance nodes as workers. As illustrated in Fig. 5, for the small data case, we observe that 4.8× speedup can be achieved by applying Lachesis; and more importantly, for the large data case, our proposed approach can achieve 14.7× speedup. Obviously, the persistent partitioning is more effective for workflows that involve larger scale of data transfer.

6.2.2 Page Rank Analytics Workflow

In the page rank 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 member of neighbours, which is 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, which is a double value. we set the

---

5The Lachesis source code can be found in https://github.com/asucactus/pangea

6Reddit datasets are download from http://files.pushshift.io/reddit/.
number of iterations to five by default, and use the default damping factor 0.85.

We benchmark the page rank application in the ten-worker cluster with high-speed-network. The producer randomly generates 40 millions to 400 millions of Page objects, with each Page object has five neighbors in average. The results are illustrated in Fig. We observe that Lachesis can achieve $1.9 \times$ to $8.0 \times$ speedups with increasing number of Page objects.

![Figure 6: Average latency of each Page Rank iteration in the high-network-speed 10-workers cluster.](image)

6.2.3 Linear Algebra Applications

We also benchmark four linear algebra applications as described in Sec.6.1. Dense matrix multiplication, Sparse matrix multiplication, gram matrix, and linear regression.

**Dense matrix multiplication.** In this workflow, the producer workload stores dense matrix as a Set of 1000 by 1000 matrix blocks which can be dispatched to a cluster of workers. Each matrix block object contains the position of the block in the large matrix specified by the row id and column id, the dimensions of the block, and the vector of values hold by this block. Then the consumer workload performs a distributed matrix multiplication in following steps:

1. It runs a join operation that pairs each matrix block having column index $j$ in the left hand matrix with any matrix block in the right hand matrix, of which the row index $i$ satisfying $i == j$. Then for each pair of joined matrix blocks, the join projection function runs to invoke the Intel MKL matrix multiplication procedure to create a 1000 by 1000 matrix block.
2. It is then followed by an aggregate operation that invokes the Intel MKL procedure to add up all 1000 by 1000 matrix blocks obtained in the above step.

In each test we create two matrices, one is 1000 rows, and varying number of columns, which is denoted as $x$, and the other matrix has $x$ rows and 1000 columns.

We first run the matrix multiplication operator with $x$ increasing from one million to ten millions, in the ten-worker cluster with high-speed network, as illustrated in Fig. 7. We observe that the performance speedups achieved by applying Lachesis increase fast with the increase in $x$, from $1.5\times$ to $2.5\times$.

We also run the workload in the ten-worker cluster with low-speed network and achieve up to $2.2\times$ speedup when increasing $x$ from one million to ten millions, as illustrated in Fig. 7.

Besides, we also run the matrix multiplication operator with $x$ increasing from one million to five millions, in the five-worker high-speed-network cluster, and also observe similar trend and up to $2.5x$ performance gain.

**Sparse matrix multiplication.** Similarly, the producer workload stores a sparse matrix as a Set of 1000 by 1000 sparse matrix blocks represented in compressed sparse row (CSR) format, in which each matrix block is represented as three arrays:

The first array holds all the nonzero entries in the block in left-to-right and top-to-bottom ("row-major") order; the second array stores the index into the first array for the first nonzero element in each row of the matrix block; and the third array contains the column index in the matrix block of each element of the first array.

![Figure 7: Performance comparison of dense matrix multiplication.](image)

The consuming workload performs the sparse matrix multiplication process, which is similar with the dense matrix multiplication, except that we use the Intel MKL sparse matrix multiplication and sparse matrix addition in the join projection function and aggregate function respectively.

In each test we also create two matrices, one is 1000 rows, and $x$ columns, and the other matrix has $x$ rows and 1000 columns.

We first run the sparse matrix multiplication operator with $x$ increasing from five millions to 30 millions, with sparsity = 0.001, in an eight-worker GCP cluster with high-speed network, and a ten-worker AWS cluster with high-speed network as illustrated in Fig. 8(a). Similar with dense matrix multiplication, we observe that the performance speedups achieved by applying Lachesis increase fast with the increase in $x$, from $1.2\times$ to $1.4\times$ in the GCP cluster, and from $1.4\times$ to $1.7\times$ in the AWS cluster.

We then run the sparse matrix multiplication operator with $x$ increasing from 40 millions to 160 millions, with sparsity $\approx 0.000001$, in the same cluster, and also observe significantly larger performance speedup, from $1.4\times$ to $3.1\times$ speedup in the GCP cluster, and $1.7\times$ to $3.3\times$ speedup in the AWS cluster, as illustrated in Fig. 8(b).

**More complex linear algebra workloads.** Based on the same producer workload for dense matrix multiplication, we also tested two more consuming linear algebra computations: gram matrix, and linear regression, as described in Sec. 6.1. Both of these two workloads involve a dense matrix multiplication operation. Besides, the gram matrix has one matrix transpose operation, and the linear
regression workload is more complex, and bottleneck-ed by the matrix inversion operation.

We test both workloads in a ten-worker cluster with high-speed network and a ten-worker cluster with low-speed network. As illustrated in Fig. 2, the gram matrix workload can achieve 1.7× speedup in the cluster with high-speed network, and 1.8× in the cluster with low-speed network. The linear regression workload can achieve 1.2× speedup in the cluster with high-speed network, and 1.5× in the cluster with low-speed network.

6.2.4 TPC-H Queries

We first test Lachesis with TPC-H scale-10 benchmark data in the high-network-speed six-node AWS cluster mentioned in Sec. 6.1. As illustrated in Fig. 10(a), we observe that Lachesis can significantly improve the performance of Q02, Q04, Q17 in this environment by 1.4×, 1.9×, and 1.7× respectively.

![Figure 9: Performance comparison of linear algebra workloads. All tests use a 1,000,000 × 1000 matrix as input.](image)

In this section, we compare the performance of a baseline system to the performance of the same system with Lachesis enabled. In various distributed settings and hardware platforms, we observe significant performance gain brought by Lachesis: up to 14 times speedup for UDF-centric data integration workflows, up to eight times speedup for analytics workflows like page rank, up to three times speedup for linear algebra workloads, and up to two times speedup for relational analytics queries. Compared to linear algebra computations and relational queries, automatic persistent partitioning is more helpful for UDF-centric applications like data integration and analytics workflows.

6.3 Overhead Analysis

In Lachesis, the overheads can be divided into two parts: the offline part that can be amortized to all data storage and partitioning requests, and the online part that is required for each store/partition request to/from the historical workflow information manager. In this section, we measure and analyze these two parts of overheads.

The offline overheads include creating signatures for historical IR graphs, and creating a super graph from historical workflow graphs. Such overheads are sensitive to number of workflows, number of workflows in each workload, and number of operations in each workload, and so on. Our measurement of offline overheads is based on the workflow statistics collected by the publicly available Workflow Trace Archive (WTA) [47] for real-world production environment, including the cloud traces collected from Two Sigma, Alibaba, Google, Shell, Pegasus and so on. As illustrated in Tab. 2, we find that the measured offline overhead of constructing super graphs and creating signatures in one r4.2xlarge instance for real-world workflows is merely up to 14 minutes, which is fully parallelizable and can be further accelerated by using multiple machines.

Table 2: Offline overhead for real-world traces. We follow the trace source name given by WTA. WF represents the number of workflows, T represents the number of tasks in a workflow, SG-latency denotes the latency required for constructing the super-graph, and SN-latency denotes the latency spent in creating signatures for IR graphs. (Latency unit: milliseconds)

| TraceName     | WF  | T     | SG-latency | SN-latency |
|---------------|-----|-------|------------|------------|
| S1. Askalon Old | 4583| 167,667 | 60 | 65         |
| S2. Askalon New | 1,835| 91,599 | 26 | 33         |
| S3. LANL       | 1,988,397| 475,355,927 | 25 | 12,638     |
| S4. Pegasus    | 56  | 10,373 | 4  | 190        |
| S5. Shell      | 3,403| 10,208 | 39 | 17         |
| S6. SPEC       | 400 | 28,506 | 12 | 25         |
| S7. Two Sigma  | 41,607,237 | 50,518,481 | 717 | 3,475     |
| S8. WorkflowHub | 10  | 14,275 | 1  | 14         |
| S9. Alibaba    | 4,210,365 | 1,356,691,136 | 94 | 39,311    |
| S10. Google    | 494,179| 17,810,002|8,861 | 835       |

In addition, the overhead of training the DRL model is another part of offline overhead, which we will describe in detail in Sec. 6.4.

We then test the Lachesis with TPC-H scale-40 benchmark data in the eleven-node AWS cluster with high-speed-network, which is also mentioned in Sec. 6.1. To generate TPC-H scale-40 data, we simply replicate TPC-H scale-10 data for four times. The results are shown in Fig. 10(b). We observe that Lachesis can also significantly improve the performance of Q02, Q04, Q17, but with slightly different performance speeds, which are 1.3×, 1.9×, and 1.9× respectively. In addition, we also observe speedup for other queries like Q12, Q13 and Q22, which is not observed in smaller cluster with smaller data size.

6.2.5 Summary

![Figure 10: Performance comparison of TPC-H queries.](image)
The online overhead for consumer at each request of computation mainly involves the matching of the running application’s IR to the partitioners associated with the input datasets to decide whether to avoid the shuffling stage. We measure the total online overhead at the consumer’s side by comparing the latency of enabling Lachesis when not partitioning any data, and simply disabling Lachesis. The results for the matrix multiplication and page rank running with five iterations are illustrated in Fig. 11. The measured time is smaller than one second for both workloads.

6.3.1 Summary

In this section, we measure various overhead incurred by the Lachesis system, including the offline overhead can be amortized to multiple data storage requests, and the online overhead at the producer’s side and at the consumer’s side. 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 online overhead at the consumer workloads’ side is negligible. The net performance gain will be further enlarged according to the write-once-read-many assumption that we mentioned in Sec 2.4.

6.4 Training Effectiveness

We train Lachesis in the five-worker AWS cluster with high-speed network, and the ten-worker AWS cluster with low-speed network. We merely use three queries in TPC-H workload: Q01, Q02, Q04 to generate training traces.

6.5 Impact of History Collection

For the PageRank workflow, we test how executing it with different sizes of input data in increasing order to form varying amount of history will affect the performance. Fig. 13 shows that if there is no historical execution of PageRank, the performance is the worst. However, if there is at least one execution of the workload, even if the historical execution is for a different input data size, the performance will get optimized similarly. This proves that our feature vector formulation for partitioner candidate is effective and data independent.

In addition, for the linear algebra workflows, we find that many different workloads (e.g. dense matrix multiplication and gram matrix) share the same desired partitioner, in this case, even if one workload has no historical execution, the system is still able to recommend the right partitioner candidate if any of other partitioning-alike workflows get executed.

6.6 Issues with Data Partitioning on Spark

We have attempted to manually create persistent partitioning for Spark applications, but that seems only possible for applications that process Hive tables, by using the bucketBy operators. However it is often difficult for UDF-centric analytics tasks to represent data as Hive tables.

We also attempt to compare the persistent partitioning in Lachesis with the intra-application partitioning in Spark for the iterative PageRank workflows in the same small-scale cluster. We run PageRank with different number of pages and links per page. Each run is configured with five PageRank iterations, and we only compare the speedup brought by partitioning mainly because it is hard to directly compare the latency of the two systems due to their complexity. As illustrated in Fig. 14, Spark partitioning can only achieve up to 1.25x speedup. The relatively lower speedup of Spark is because of two reasons: first, the persistent partitioning doesn’t require any shuffling at application runtime, but intra-application partitioning require one-time online partitioning overhead that is amortized over the five iterations; second, the CPU-intensive nature of Java-based systems like Spark indicates that the shuffling overhead may not be the major performance bottleneck.
Intermediate Representation for UDF-centric Analytics. Some systems such as TupleWare [12] compile user application code into IR at AST level. However, this is unlikely to work in every case. It is difficult or impossible to reason about the behavior of user defined functions that encompass hundreds or thousands of lines of domain-specific code for operating over arbitrary objects. An alternative is to use a DSL rather than an opaque UDF to customize relational operators. The DSL can force the programmer to expose intent. This is the approach taken by SparkSQL [4], for example. Using the DSL, Spark can analyze the AST of the code and understand the semantics of an expression such as employee1.startYear === employee2.endYear + 1. However, after such an AST tree has been optimized, it is directly converted into Java byte code and the semantics are lost and a code snippet like employee2.endYear + 1 can not be extracted separately and re-invoked in the future. This means that it cannot be used to perform workload-based optimization. Also SparkSQL SQL can not work with highly nested objects and arbitrary UDFs as to our knowledge. Weld IR [14] and Lara IR [24] map python code to a high-level functional language. However, similar with SparkSQL DSL, the IR and the code generation are fully decoupled. This means if you want to reuse the code for some part of the IR, e.g. a function that maps an object to a key, you need first extract the IR fragment, modify it into a complete Weld IR program, and generate code from it using a compiler. This process is difficult to be automated.

All of these existing DSL/IRs have a potential to be used for persistent partitioning, however, we argue that our proposed lambda calculus IR is designed to facilitate sub-computation extraction and reuse, which is a critical enabler for efficient automatic partitioning.

8. CONCLUSION

In this paper, we argue that automatically creating persistent data partitionings for Big Data applications is an important and challenging task, particularly for UDF-centric workloads. We propose Lachesis to address the problem, which includes a unique DSL and IR that can facilitate the analysis, extraction, reuse, and matching of sub-computations in UDFs. Lachesis also provides a data placement optimizer based on deep reinforcement learning approach and historical workflow analysis. The evaluation results demonstrate that Lachesis can bring up to $14 \times$ performance speedup for various Big Data integration and analytics applications, and can be effective with different data sizes and different environments. In the meanwhile, it costs a few hours’ model training overhead and less than 15 minutes offline overhead for constructing the supergraph from historical workflows, both of which can be amortized to many data storage requests. In addition, it incurs up to 10% online overhead at the producer’s side which can be amortized by multiple consuming workloads, and negligible online overhead at the consumer’s side.

9. ACKNOWLEDGEMENTS

Thanks to two ASU graduate students: Pradeep Jampani for helping with the implementation and benchmark of the sparse matrix multiplication, and Srihari Jayakumar for helping with the benchmark of the dense matrix multiplication. Thanks to Don-paul Stephens for his insightful feedbacks on an earlier version of this paper. Most of the work presented in this paper has been supported by the ASU faculty start-up funding, AWS Cloud Credits for Research program, Google GCP research credits program, and DARPA MUSE award No. FA8750-14-2-0270.
A. MORE DSL/IR EXAMPLES

A.1 Three-way Join

In this example, we join three Reddit datasets: (1) Reddit submissions, which is a 6 terabytes’ JSON dataset including all reddit posts submitted in Aug 2019; (2) Reddit accounts, which is 1.2 terabytes’ CSV dataset containing the information about 78 millions of reddit users; and (3) the subreddits dataset, which is a 1.6 terabytes’ JSON dataset including the information about all reddit forums. The goal is to form a training dataset for automatically recommending posts to reddit users. We show the join filter function $f_{fil}$ expressed in lambda calculus in Listing 4. We omit the project function $f_{proj}$ of this join operator here.

Listing 4: Reddit join: $f_{fil}$ expressed in lambda calculus DSL

```
lambda<bool> join_filter(string submission, string account, string subreddit) {
    return (func(submission, json_parse)[“author”] == func(account, csv_parse)[1] && func(submission, json_parse)[“subreddit_id”] == func(subddit, json_parse)[“id”]);
}
```

The above expression will return a tree of lambda terms as illustrated in Fig.15. The logical query optimizer will traverse the tree of lambda terms, extract partitioner candidates as a subgraph in the IR, and store the IR as a persistent PC object [58], as well as the index of each partitioner candidate into the Lachesis, while applying other optimization techniques, such as removing the redundant lambda term, fuse operators and so on.

Listing 5 shows the code of using the binary join operator in dataflow-based DSL [34,52] to implement the logic of Listing 4. You will see it depends on the programmers to determine the join ordering, whether to first join submissions with accounts, or first join submissions with subreddits, thus it is not declarative. In contrary, if our proposed DSL is used as in Listing 4, the join ordering can be automatically determined by traversing the IR via a query optimizer.

Listing 5: Reddit join: expressed in dataflow-based DSL

```
intermediate_pairs = submission_pairs.join(account_pairs).map((author, ((subreddit, submission), account)) => (subreddit, (submission, account)))
//then join with subreddits
subreddit_pairs = subreddits.map(subreddit =>
  (json_parse(subreddit)[“id”])
}
result = subreddit_pairs.join(intermediate_pairs)
```

Figure 15: IR fragments for the 3-way reddit datasets join.

A.2 Log Merge

In this example, we want to develop a universal tool to merge-sort arbitrary set of large log files into one dataset that is ordered by timestamp [28]. Each log file may have hundreds of gigabytes to terabytes in size, and may use different timestamp formats, such as ISO-8610 standard, cloud-init standard, Unix time, and so on. So the sort key extraction function ($f_{keyProj}$) needs to match multiple regex expressions. Its implementation in our proposed DSL is illustrated in Listing 6.

Listing 6: Merge-Sort: $f_{keyProj}$ expressed in lambda calculus DSL

```
lambda<DateSort> keyProj(string logrec) {
    lambda<Match> match = func(logrec, iso8601_patternmatch);
    if (member(match, res))
        return member(match, datetime);
    lambda<Match> match = func(logrec, cloudinit_patternatch);
    if (member(match, res))
        return member(match, datatime);
    return member(func(logrec, unix_patternmatch), datetime);
}
```

The corresponding IR is illustrated in Fig.16. It is obvious that the whole $keyProj$ function ($f_{keyProj}$) of the sort operator can serve as the partitioner candidate to extract partition keys from log records for range partitioning. This indicates that the user can simply put all of the projection logic into one opaque lambda term because there is no need to decompose the sort key extraction logic in this case.

Figure 16: IR example for the sort key projection function, which also serves as a partitioner candidate for the log files. (Solid line represents the dataflow and dashed line represents the control flow.)
A.3 Similarity Join

It is possible that the input data may be transformed through a series of operators before it is being joined. For example, assume we need detect Reddit posts submitted in July 2019 that are similar to Reddit posts submitted in Aug 2019, a similarity join between the July 2019 dataset and the Aug 2019 dataset can solve the problem. The application consists of three operators: two flatten operators (multiselect) that derives multiple locality sensitive hash (LSH) signatures from each post for both datasets, followed by a join operator that returns all pairs of Aug post and July post that share at least one LSH. The projection UDF $f_{proj}$ of the multiselect operator and the filter UDF $f_{filt}$ of the join operator are illustrated in Listing 8 and Listing 9 respectively.

Listing 8: $f_{proj}$ of multiselect in lambda calculus DSL

```lambda<vector<pair<string, string>>> proj(string submission) {
    //the multi_probe is a UDF invoked from a LSH library, it returns a list of strings as LSH signatures.
    return func(submission, multi_probe);
}
```

Listing 9: $f_{filt}$ of join expressed in lambda calculus DSL

```lambda<bool> join_filter(pair<string, string> lsh_aug_sub_pair, pair<string, string> lsh_jul_sub_pair) {
    return lsh_aug_sub_pair[0] == lsh_jul_sub_pair[0];
}
```

The end-to-end IR of the application is illustrated in Fig. 17. Different with other two examples, the partitioner candidates extracted from this application span the lambda terms of multiple operators.

Figure 17: IR for the Similarity Join example.

2. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. Proceedings of the VLDB Endowment, 2(1):922–933, 2009.
[2] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In Proceedings of the 2004 ACM SIGMOD international conference on Management of data, pages 359–370. ACM, 2004.
[3] A. Alexandrov and et al. The stratosphere platform for big data analytics. VLDB, 23(6):939–964, 2014.
[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pages 1383–1394. ACM, 2015.
[5] L. Babai. Graph isomorphism in quasipolynomial time. In Proceedings of the forty-eighth annual ACM symposium on Theory of Computing, pages 684–697, 2016.
[6] H. P. Barendregt et al. The lambda calculus, volume 3. North-Holland Amsterdam, 1984.
[7] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, et al. Systemml: Declarative machine learning on spark. Proceedings of the VLDB Endowment, 9(13):1425–1436, 2016.
[8] W. Chen, M. Kifer, and D. S. Warren. Hilog: A foundation for higher-order logic programming. The Journal of Logic Programming, 15(3):187–230, 1993.
[9] Y. Chen and et al. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. VLDB, 5(12):1802–1813, 2012.
[10] Z. Chen, Y. Wang, V. Narasayya, and S. Chaudhuri. Customizable and scalable fuzzy join for big data. Proceedings of the VLDB Endowment, 12(12):2106–2117, 2019.
[11] T. P. P. Council. Tpc-h benchmark specification. Published at http://www.tpc.org/tpcouncil. html, 21:592–603, 2008.
[12] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Cetintemel, and S. B. Zalonik. Tupleware: “big” data, big analytics, small clusters. In CIDR, 2015.
[13] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. Proceedings of the VLDB Endowment, 3(1-2):48–57, 2010.
[14] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). Proceedings of the VLDB Endowment, 3(1-2):515–529, 2010.
[15] G. Eadon, E. I. Chong, S. Shankar, A. Raghavan, J. Srinivasan, and S. Das. Supporting table partitioning by reference in oracle. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 1111–1122, 2008.
[16] M. Y. Eltabakh, Y. Tian, F. Ozcan, R. Gemulla, A. Krettek, and J. McPherson. Cohadoop: flexible data placement and its exploitation in hadoop. Proceedings of the VLDB Endowment, 4(9):575–585, 2011.
[17] P. Francisco et al. The netezza data appliance architecture: A platform for high performance data warehousing and analytics, 2011.
[18] B. Hilprecht, C. Binnig, and U. Roehm. Learning a partitioning advisor with deep reinforcement learning. arXiv preprint arXiv:1904.01279, 2019.
[19] B. Hilprecht, C. Binnig, and U. Röhm. Learning a partitioning advisor for cloud databases. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pages 143–157, 2020.
[20] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. Computation reuse in analytics job service at microsoft. In Proceedings of the 2018 International Conference on Management of Data, pages 191–203. ACM, 2018.
[56] J. Zhou, N. Bruno, and W. Lin. Advanced partitioning techniques for massively distributed computation. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pages 13–24. ACM, 2012.

[57] J. Zhou, P.-A. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010), pages 1060–1071. IEEE, 2010.

[58] J. Zou, R. M. Barnett, T. Lorido-Botran, S. Luo, C. Monroy, S. Sikdar, K. Teymourian, B. Yuan, and C. Jermaine. Plinycompute: A platform for high-performance, distributed, data-intensive tool development. In Proceedings of the 2018 International Conference on Management of Data, pages 1189–1204. ACM, 2018.

[59] J. Zou, A. Iyengar, and C. Jermaine. Pangea: monolithic distributed storage for data analytics. Proceedings of the VLDB Endowment, 12(6):681–694, 2019.

[60] J. Zou, A. Iyengar, and C. Jermaine. Architecture of a distributed storage that combines file system, memory and computation in a single layer. The VLDB Journal, pages 1–25, 2020.