Multi-tenant Federated Learning

David Weissteiner

Bachelor's Thesis

to achieve the university degree of
Bachelor of Science

submitted to
Graz University of Technology

Supervisors
Univ.-Prof. Dr.-Ing. Matthias Boehm
M.Tech Arnab Phani

Institute for Interactive Systems and Data Science

Graz, August 2022

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present bachelor's thesis.

Date

Signature

Abstract

Federated Learning (FL) is being increasingly studied and became a common application in many domains. Existing work about FL mostly focuses on a system architecture with a single centralized actor (coordinator) deploying continuous computation requests over multiple decentralized federated workers. However, due to the limitation of a single coordinator within the federated infrastructure, only one actor can address the data of the federated worker at the same time.

The emerging field of FL created many application scenarios allowing to learn centralized models from private data, and some systems even making it feasible to perform certain ML pipelines. However, these applications miss the benefits of multiple coordinators within the same federated infrastructure.

In this work, we introduce Multi-tenant Federated Learning (MuTeFL), which provides the possibility to share data and computing resources of multiple federated workers among several distinct coordinators. To preserve the independence of the coordinators, we create an autonomous, server-like federated worker that can serve multiple coordinators simultaneously. Moreover, we isolate the coordinator-instances at the federated worker to avoid revealing data to other coordinators, we introduce parallelization strategies to utilize the full potential of the worker, and we eliminate redundancies at the worker by using reuse techniques.

We thereby create the possibility to deploy the federated worker as a long-running server process that can be addressed by different parties at any time. This enables, for example, simultaneous model training by different data scientists on the same federated workers, as well as providing learning capabilities on sensitive data to the public domain.

Contents

1 Introduction

The potential of machine learning (ML) applications is expanding rapidly as data collection capabilities increase. Since collecting user data becomes steadily easier due to new technological developments, an increasing amount of data is being used to train machine learning models to make future decisions about user behavior. In today's world, nearly everyone carries a smartphone with them that is capable of gathering various pieces of information through built-in sensors (e.g., location, movement) as well as through the use of the device (e.g., keyboard inputs, calls). Since most of this data contains sensitive information that users do not want to disclose, the challenges of protecting data privacy and preserving data ownership arise. Additionally, a major bottleneck in geo-distributed applications is created by the transmission of entire datasets over the network. Sometimes it is even infeasible to consolidate the data in a centralized way due to bandwidth limitations (e.g., collecting detailed data from data centers around the world) [55].

Federated Learning (FL) is a sub-field of ML which enables model training on decentralized data utilizing the computing resources of the corresponding decentralized nodes (federated workers), and addresses problems of privacy, ownership, and the locality of data [12]. Privacy of federated data is achieved by only allowing certain aggregates of the data to be sent to the centralized node (control program), in order to prevent data reconstruction and to conceal sensitive statistics of the dataset. Beyond data privacy preservation, training models at the federated workers usually reduces the performance bottleneck from sending large data files over the network, since the data is aggregated on-site and is transferred to the control program as smaller-sized aggregates.

1.1 Motivation

During the last years, FL techniques experienced rapid growth in their applications, which cover a broad spectrum of application domains and fields like mobile devices, industrial engineering, and health care [35]. As privacy preservation emerges as a key advantage of federated learning, a great number of cross-device systems have evolved towards FL strategies to protect sensitive data [15, 49]. However, also the aspect of data locality led to great developments in the industry, as this allows data scientists to analyze data without the overhead of central data consolidation. Although the majority of today's FL applications are dealing with company-internal learning scenarios (i.e., within the company itself or the company's product), also inter-company use-cases have been developed to train models on sensitive data from different parties without revealing the actual data.

Existing work on Federated Learning focuses on a single-tenant-setup, where the federated workers are designed to execute received commands from a single control program processing a data science pipeline. Therefore, all parts of an FL network are built ad-hoc for a single learning procedure.

Example 1. *Consider a company with a large data science team. Every data scientist analyzes data from various federated data sources on their individual business laptops. Since multiple data scientists analyze the same data—or data at the same federated site—some of the federated workers need to be shared among them.*

In common scenarios as in Example 1, data scientists have to make explicit agreements about separate usage times of the federated workers. Automated Federated Learning systems, on the other hand, often implicitly alternate their usage, as for example the Federated Learning in Gboard (the Google Keyboard), which engages devices in model training only when they are idle [38].

Multi-Tenancy: We introduce Multi-tenant Federated Learning (MuTeFL) to overcome these limitations of a single processing instance by allowing multiple control programs to concurrently train models on the same federated data. This results in autonomously running federated workers and addresses the limitation of workers being bound to one single coordinator. Applied to Example 1, there is no need for an agreement about different usage times anymore, as the data scientists can work simultaneously on the same federated workers.

Computational Redundancy: The typical data science process includes data pre-processing and data analysis. Often, applied pre-processing methods are highly data-dependent. With multiple data scientists performing machine learning on the same federated site, we identify the repetitive data pre-processing step as an unnecessary redundancy. Furthermore, since data scientists often perform ML pipelines iteratively making only small changes (for example in hyper-parameter selection), multiple operations within the pipeline are independent of the changes made, and hence are redundantly computed several times.

1.2 Contributions

In this thesis, we describe the overall architecture and key concepts for extending a federated backend to support multi-tenant execution, and share insights on how we support Multi-tenant Federated Learning (MuTeFL) in Apache SystemDS [10]. Additionally, we show necessary changes for sustaining robustness and performance, and demonstrate some optimizations to enhance the adapted system architecture.

With MuTeFL we create the possibility for a multi-tenant-setup of the FL infrastructure as shown in Figure 1.1. Complete isolation of the coordinator-specific operations and intermediate results at the federated worker—in terms of the separation of their processing contexts—allows for multiple independent data scientists to perform individual learning procedures concurrently on the same shared federated data, and hence on the same federated workers. Furthermore, such a complete isolation of the contexts opens the opportunity to run the federated workers as stand-alone servers.

Subsequently, we concentrate on the efficient utilization of the resources from the federated worker. For that reason, we introduce parallelization strategies considering multi-tenant execution and the difference in computing resources of the nodes in a federated infrastructure.

We then optimize the adapted system with a particular focus on emerging redundancy between the differentiated processing contexts. Therefore, we apply reuse strategies to intermediates from redundant operations, whereby we eliminate some time-consuming computations.

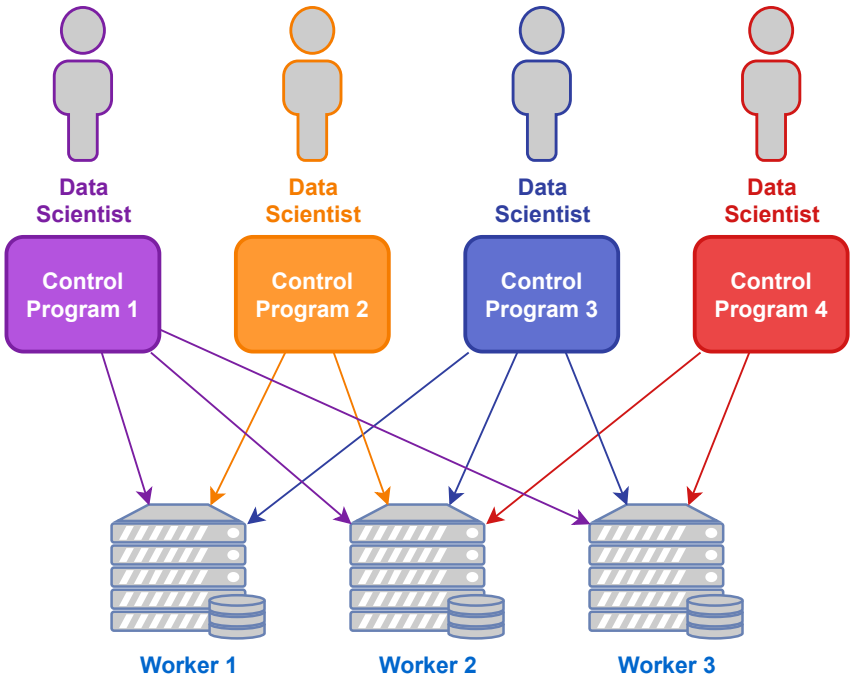

Figure 1.1: Multi-tenant Setup

Finally, we present results from an extensive evaluation to show the robustness and efficiency of the adapted system. Moreover, we show the impact of the techniques introduced in this thesis by conducting an ablation study regarding individual techniques. Thereby, we explore the advantages of each individual extension.

Our technical contributions are:

1. *Tenant-Identification*: the identification and distinction of the different control programs at the federated worker (Section 3.1)
2. *Tenant-Isolation*: the isolation of processing instances through a complete separation of the processing contexts at the federated worker (Section 3.2)
3. *Multi-threaded Processing*: the multi-threaded execution of federated requests and operations at the federated worker (Section 3.3)
4. *Optimizations*: caching and reuse of common intermediates within and across tenant sessions (Section 3.4-3.8)
5. *Experiments*: experimentally evaluate the robustness and the capabilities of the newly adapted system (Section 4)

2 Background and Challenges

In this chapter, we provide an overview of the existing federated backend of Apache SystemDS [10] (formerly Apache SystemML [11]), and characterize major challenges that show up when extending the system to support multi-tenant execution. Additionally, we analyze the necessary changes in order to find possible optimizations of the adapted system.

2.1 SystemDS Federated Runtime

Apache SystemDS is a machine learning system that perfectly qualifies for extending it to support Multi-tenant Federated Learning. Recent work within the ExDRa project has introduced a federated runtime backend to SystemDS, which provides a great number of federated primitives to allow the automatic compilation of machine learning algorithms into federated runtime plans [9].

The basic federated system configuration of SystemDS is characterized by three important components:

1. The *Main Control Program* working with federated data objects (Section 2.1.2) as the initiating and driving component of the federated learning process.
2. The *Federated Workers* as decentralized computing nodes that own data and process operations on these data (described in Section 2.1.1).
3. The *Network Communication* as the interface between the main control program and the federated workers.

As a general baseline throughout this section, we consider the usual setup of a federated learning infrastructure, where the main control program serves as a coordinator for the federated workers. In this scenario, the coordinator trains models on decentralized private data that is located at the federated sites.

2.1.1 Federated Workers

From the coordinators' point of view, a federated worker is a network node that represents the coupling of decentralized data objects with a computing unit. As the coordinator runs machine learning scripts on federated data objects, it compiles runtime plans that contain federated operations and directs the workers to perform the respective compiled operations.

Federated Workers are designed to carry out commands from the coordinator, providing computing capabilities over their respective decentralized data. In fact, a worker is the same program as the coordinator but started in a server-like mode listening to a specific port at the federated site. Generally, the federated worker waits passively for requests from the coordinator, performs the required computations which might change the currently live intermediates of the worker, and responds with the appropriate information. However, to simplify the interface between the coordinator and the worker, and to reuse operation kernels, the possible requests are limited to six predefined general subroutines, described in Section 2.1.3.

2.1.2 Federated Data Objects

The term Federated Data denotes meta data at the coordinator, pointing to subsets of data at the individual federated workers. Usually, federated learning addresses the composition of several data parts at the federated workers as a single partitioned federated data object. Although there are no limitations on the partition scheme, two scheme types emerge as predominant in practice: *row-partitioned* (also horizontal FL) and *column-partitioned* (also vertical FL).

In SystemDS, a federated data object at the coordinator comprises multiple arbitrary, virtual data partitions that point to the actual data partitions at the federated workers. The key property of a federated data partition is that it resides on the federated worker and can solely be addressed by the control program (coordinator) through predefined requests to the worker (Section 2.1.3). In order to define a federated data object, the coordinator creates a virtual data object with information about the composition of the partitions, as well as the main properties of the partitions such as the network address of the worker nodes, the partition filename, and the individual dimensions of the partitions. The following code example shows the

creation of a row-partitioned federated matrix from the coordinator's perspective in SystemDS' scripting language DML.

```
X = federated(addresses=list(node1/file1, node2/file2, node3/file3),
      ranges=list(list(0,  0),  list(15k, 80),      # node1/file1
                  list(15k,0),  list(70k, 80),      # node2/file2
                  list(70k,0),  list(110k,80)));    # node3/file3
```

2.1.3 Federated Requests

The communication between the control program (coordinator) and the federated workers is based on remote procedure calls (RPCs). The following six predefined RPC procedures (called Federated Requests) specify the interface between the coordinator and the worker:

1. *READ*: read a data object from the filesystem and store it in the symbol table
2. *PUT*: extract the data object (transferred from the coordinator within the request) and store it in the symbol table
3. *GET*: get the data object with the specified ID from the symbol table of the worker and return it as response
4. *EXEC_INST*: execute the specified instruction with the specified inputs and write the result to the specified output
5. *EXEC_UDF*: execute a given user-defined function[1] (UDF) object
6. *CLEAR*: clean up the symbol table

2.1.4 Federated Runtime Plans

Performing model training on a federated data object with multiple federated partitions often requires a coordinated consolidation of the resulting aggregates. In SystemDS, specialized implementations of federated instructions coordinate the execution of operations at the federated workers and potentially consolidate partial results. Depending on the instruction as well as the partition scheme and privacy constraints of the federated data, different types of consolidations have to be performed.

[1] UDFs are objects with a set of input IDs, a set of output IDs, and an arbitrary execute method to perform computations with the specified inputs and outputs

When executing Federated ML pipelines, the coordinator takes the responsibility to control the actions of the respective federated workers. In order to achieve this, a compilation of federated runtime plans is required. Therefore, with the essential federated plan compilation in SystemDS, the coordinator first creates a basic runtime plan of local or distributed Spark instructions. In a second step, instructions on federated data are replaced by the corresponding federated instructions. Besides the translation from local instructions into federated instructions, SystemDS supports cost-based and heuristic planners to directly compile the program into federated instructions.

2.2 Multi-Tenancy

Adapting an FL system to support multi-tenant execution faces several challenges. In this section, we describe the major problems of such an extension and elaborate on their origin.

2.2.1 Tenant Isolation

The key property of a multi-tenant federated backend is, that it supports concurrent training from multiple coordinators on the same federated sites. Hence, the federated workers must be capable of processing simultaneous requests of different coordinators.

With the single-tenant design of SystemDS' federated backend, workers do not have to consider more than one sequential event sequence because only one coordinator sends queued requests in a consecutive manner. Additionally, the federated workers maintain just a single symbol table, since there cannot occur any variable name conflicts because the coordinator controls the creation and deletion of entries in the worker's symbol table.

In multi-tenant mode, workers have to consider concurrent requests from multiple coordinators. Since the coordinators control the variable creations at the worker without synchronizing with each other, the problem of variable name conflicts appears. Therefore, the challenge of isolating the symbol tables of different coordinators at the worker arises.

2.2.2 Tenant Distinction

Before the tenants can be isolated at the federated worker, the worker needs to distinguish between different tenants. As the coordinators are independent of each other and do not even know each other, it is impractical to assign an incrementing unique ID to them. Thus, the challenge arises to uniquely identify each distinct coordinator from the perspective of a federated worker.

Furthermore, tenant identification must not depend on properties that can be affected by the coordinator itself, since it should not be possible for an adversarial coordinator to imitate another coordinator. By this uninfluenceable identification we are able to ensure that a tenant cannot retrieve or modify the intermediates of other tenants, and thus prevent attacks from other coordinators.

2.2.3 Multi-Tenant Redundancy

As discussed before, we need a complete isolation of the coordinators' individual symbol tables to support multi-tenant execution at the federated site. Since the federated worker typically refers to certain federated data, we recognize a high reuse potential amongst the coordinator contexts on the federated worker.

A basic execution pipeline at the federated worker starts by reading the respective data partition from the file system. Subsequently, pre-processing steps are performed to prepare the federated data. Finally, when everything is set up, the worker performs the learning task and responds to the requesting control program with either an empty success message or with the resulting data.

Concurrent processing of multiple pipelines yields unnecessarily created intermediates and redundant computations. Consider the scenario of multiple coordinators performing the same pipeline. With completely isolated symbol tables at the federated worker, every operation gets separately computed for each coordinator and materializes the same result multiple times. In this case, the federated worker redundantly performs the whole pipeline several times—including the reads from the filesystem, all computations, and even the serialization of the resulting response.

Nevertheless, redundant computations exist even in the case of different ML pipelines. Because of the affiliation of certain federated data to a particular federated worker,

it is very likely that different pipelines have to load the same data from the filesystem. In addition, the type of data pre-processing is very data-dependent, which implies that ML pipelines on the same data usually have the same pre-processing steps. Therefore, redundancy may exist in reading, computing, and serializing individually.

For reusing across the isolated symbol tables, we have to detect identical intermediates. As computations consist of an arbitrarily large DAG (directed acyclic graph) of operations, the challenge is to identify and match equivalent computations. More precisely, we have to match the whole DAG of operations, as the entire history of a computation—including the provenance of the data objects involved—determines its identity. Since these operations often involve broadcast data from the coordinator, the additional challenge is to inform the worker about the provenance of the transferred data to deduplicate subsequent operations.

3 System Design

In this section, we provide an overview of the steps taken to support Multi-tenant Federated Learning (MuTeFL) in SystemDS. Furthermore, we elaborate on the reasons behind the design decisions made and explain how we integrated the introduced components into the system.

3.1 Tenant Distinction

As already indicated in Section 2.2.2, we need an ability to differentiate the coordinators at the federated worker. Without this capability, the worker will not be able to isolate variables that belong to different tenants.

To accomplish the separation between coordinators at the federated worker, we need to identify the source coordinator for every request in order to associate the request with the proper coordinator and execute it within the respective environment. This raises the question of what makes a coordinator unique from the worker's perspective. The worker knows the IP address of the coordinator because requests are sent over a network connection. Anyway, this alone does not necessarily identify a particular coordinator, since it is just the public IP address that could be shared by numerous coordinators located behind a proxy. Additionally, it may be desirable to run multiple coordinators on the same machine, e.g., when several data scientists are sharing a server. To provide these capabilities, we require an additional identifier to distinguish between the coordinators.

In SystemDS, every message that is transmitted from the coordinator to the federated worker opens a fresh network channel. Since a new channel also implies a changed outgoing port, we cannot rely on the source port as an identifier of the coordinator. Instead, we include the process ID of the coordinator inside the federated request and obtain it as an additional identifier at the worker. Hence, we

differentiate the coordinators based on the combination of their public IP address and their process ID.

Although the process ID resolves the problem of multiple coordinators running on the same machine, coordinators on different machines but behind the same proxy, which coincidentally have the same process ID, can still lead to a collision. As this is a very rare situation, we define and document this particular case as a known limitation of the system and do not address it further.

A second system limitation is showing up in networks using the Dynamic Host Configuration Protocol (DHCP). Since the IP address is part of the tenant's identifier, the coordinator is recognized as a new tenant if its IP address has changed. This leads to incorrect identification whenever the IP address is reassigned by DHCP. Given the fact that this problem only occurs when the IP address is renewed, and that offices usually configure a DHCP Lease Time[1] of at least 24 hours [18], this particular issue can be considered a limitation of the system.

3.2 Federated Lookup Table

In order to avoid variable collisions at the federated worker, complete isolation of the coordinators' processing contexts is required. Otherwise, the coordinators will interfere with each other at the worker, resulting in overwrites and memory corruption because they are not aware of each other's variable handling.

We resolve the issue of interfering tenants by assigning each coordinator an individual symbol table for life variables at the federated site. Then, the worker is fully responsible to handle the isolation by storing the symbol tables, linking them to the respective coordinators, and ensuring that tenants can only access intermediates of their contexts. Before processing a federated request, the worker switches to the symbol table, which corresponds to the requesting coordinator, and performs the request on top of this particular environment. In contrast to an approach where the variable ID handling is delegated to the federated worker, there is no need for the

[1] The DHCP Lease Time is the time period that an IP address is reserved for a network device. Upon expiration of the Lease Time, the IP address is returned to the allocation pool and the DHCP server can reallocate it to another device.

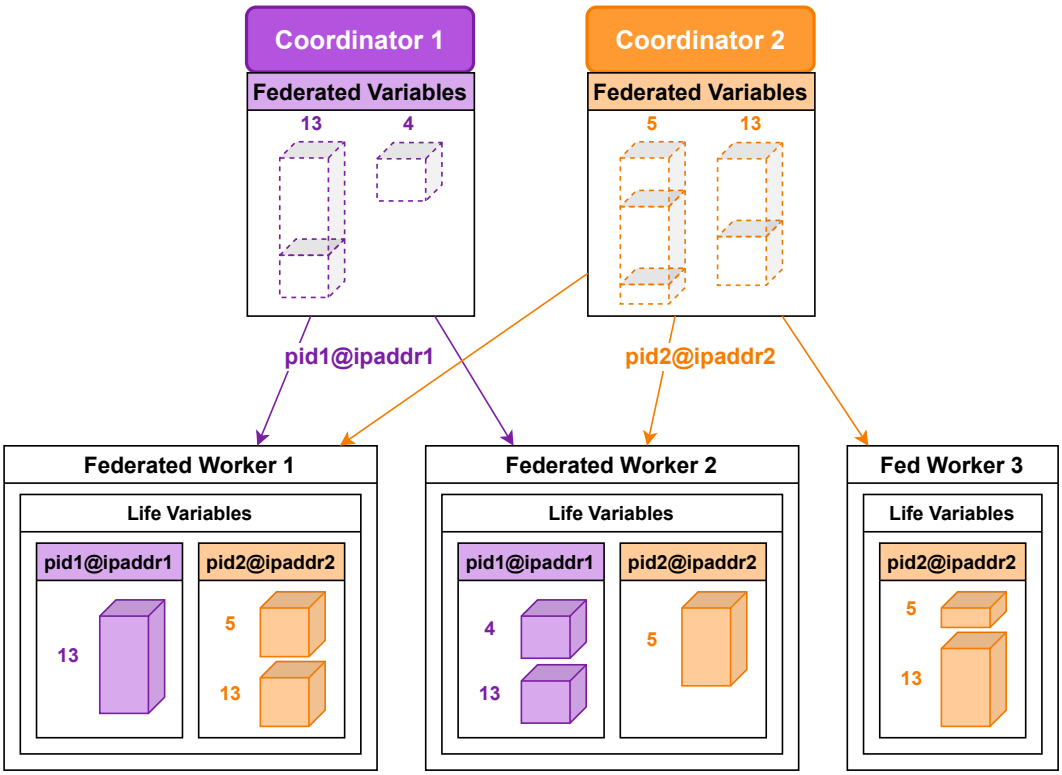

Figure 3.1: Tenant Isolation

coordinator to be aware of the isolation or anything related to it, since it is handled passively at the worker.

With tenant-specific symbol tables, every coordinator is allowed to independently control the execution process at the worker without synchronizing with others. This approach enables the coordinators to create, modify, and delete variables on the worker, regardless of whether other coordinators use the same variable ID on the worker or not (illustrated in Figure 3.1).

In SystemDS, the symbol table is stored inside a so-called execution context which comprises the essential components of a certain computation procedure. The execution context itself is enclosed by a context map that manages the creation of execution contexts. This respective manager allows running the content of a parallel for loop simultaneously by assigning an individual execution context to every

parfor worker. Thus, the concurrent loop instances are isolated from each other and cannot interfere.

When it comes to MuTeFL, we face the same problem as with the parfor loop—multiple concurrent accesses to variables that lead to collisions. Therefore, we address this challenge similar to the context distribution of the parfor loop. In order to keep the existing concept of execution context distribution, we perform the division one level above at the context map.

We introduce the Federated Lookup Table that maps from a unique coordinator identifier to a coordinator-specific execution context map. Whenever a request arrives at the federated worker, the worker identifies the coordinator as specified in Section 3.1 and loads the corresponding execution context map from the lookup table. Since the context map contains the actual contexts, every coordinator receives a separate execution environment, and hence also a unique symbol table.

By assigning separate execution contexts to each individual coordinator, we ensure that the intermediates of one coordinator are invisible to the other coordinators. Since we process requests at the federated worker only using the appropriate coordinator-specific execution context, coordinators can only address variables from their respective contexts. As shown in Figure 3.2, requests by different coordinators are processed with their respective execution contexts obtained from the federated lookup table. Furthermore, it is possible to perform parallel requests (e.g., from a parfor loop), since the context map handles the differentiation of contexts for parfor loops.

3.3 Multi-threaded Federated Worker

MuTeFL introduces the opportunity for multiple simultaneous computation requests to the worker by more than one coordinator. Therefore, on the one hand, the computations have to be performed in a concurrent manner, and on the other hand, the worker should utilize all its available resources to process the existing requests.

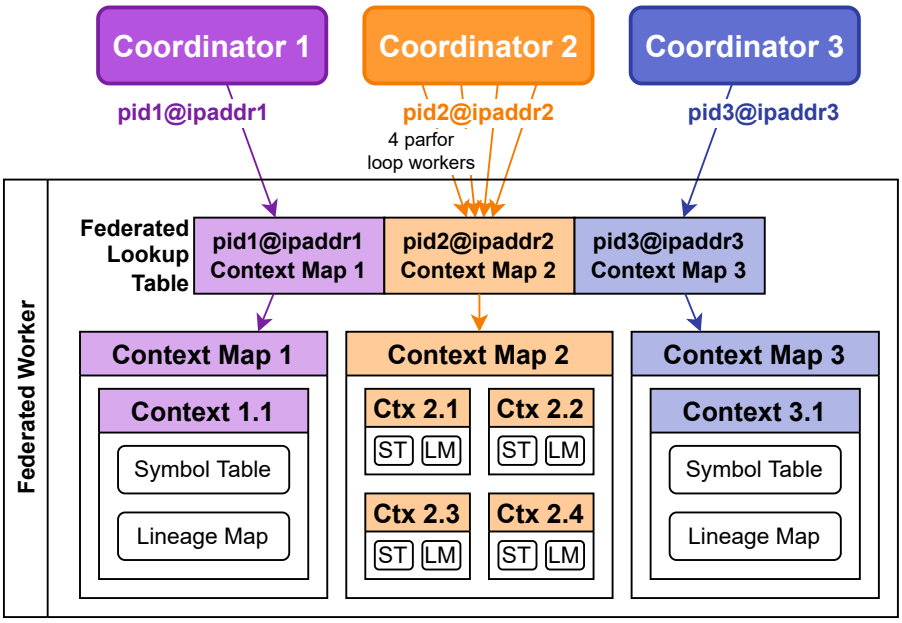

Figure 3.2: Federated Lookup Table

Request Processing

As multiple tenants would like to perform simultaneous federated computations, the federated worker needs to dispatch and process the requests concurrently. Otherwise, one coordinator would receive the entire computational capacity of the worker, while the other coordinators would have to wait for their turn. Some computations could occupy the worker for several hours (e.g., a parameter server worker thread), which constitutes a significant obstacle in practice. In addition, potentially involved I/O operations show substantial blocking time that could be more effectively leveraged. Hence, multi-threaded handling of requests is needed to establish responsiveness of the federated worker.

SystemDS makes use of Netty as a networking framework for the communication between the nodes of a federated infrastructure. On the federated site, a server socket listens to a specific port for incoming requests. Netty's event-driven dispatching of requests is based on the Reactor architectural pattern, where the mechanisms to demultiplex and dispatch events are separated from the actual application-specific process by the use of distinct thread pools, known as the boss group and worker

group [50]. When receiving a request from a coordinator, it is forwarded from the network interface to the boss event executor group, which dispatches the requests to further event executor threads from the worker group for the actual execution. The worker-threads are then responsible to carry out the request and create the response. For a single-threaded request processing, only one thread per group is required. The boss-thread accepts the requests and instructs the worker-thread to perform them, sequentially. With the single-threaded event loop, concurrent requests are queued and processed one after the other.

To enable simultaneous request processing in SystemDS, we should first determine the duties of the boss group and worker group. The only task of the boss-thread is to accept the incoming serialized federated requests and distribute them to the available worker-threads or queue them if no worker-thread is available at the moment. As this distribution task is not very computing intensive, we identify the possibility to configure the boss-group single-threaded. The time-consuming computations (i.e., the request deserialization, the actual request processing, and the response serialization) take place at the level of the worker group. Therefore, a multi-threaded configuration of the respective worker event executor pool becomes necessary to eliminate the issue of long-lasting, blocking computations at the federated site.

Anyhow, the thread pool of the worker group has to be limited by an upper bound, since creating too many threads would result in a decrease in performance. A general state-of-the-art guideline is to relate the thread pool size to the number of CPU cores. Furthermore, the book *Java Concurrency in Practice* [26] suggests configuring the pool size in accordance with the following formula:

$$N_{threads} = N_{cpu} \cdot U_{cpu} \cdot \left(1 + \frac{W}{C}\right) \tag{3.1}$$

This formula relates the number of available CPU cores N_{cpu} to the desired factor of CPU utilization U_{cpu} and scales it by the expected ratio of wait time W to compute time C. In our case, we aim to utilize the full available CPU resources at the federated site. Hence, the target CPU utilization is $U_{cpu} = 1$. The ratio of the wait time to the compute time is mostly affected by executed I/O operations. Since the received federated requests can consist of I/O operations as well as purely computational operations, we cannot stick to a general value for this ratio. However, taking into consideration that the upper bound of the worker threads is applied

only to the worker group, and that the underlying operations have their own level of parallelism, we decide to set this ratio to the smallest value $\frac{W}{C} = 0$. Therefore, we cap the maximum number of threads for the worker group by the number of CPU cores $N_{threads} = N_{cpu}$.

Operation Execution

As already mentioned in the previous section about Request Processing, we aim to utilize the entire computation capacity for the federated worker. Therefore, we set the concurrency level for request processing according to the available number of CPU cores. However, this type of concurrency only applies to simultaneously processed federated requests from different coordinators. Hence, if only a single coordinator makes use of the federated worker, the capacity of the worker is not fully utilized.

Since operation-level multi-threading also has major importance in local execution with SystemDS, most of the operators are already qualified to run with multiple threads. As indicated in Section 2.1.4, we create the runtime plan at the coordinator and request the federated worker to execute the contained federated instructions. These instructions, however, were constructed with the level of parallelism of the coordinator. Therefore, also the included operator is created with the coordinator-specific number of threads. In the case of different CPU capacities of the coordinator and the workers, the worker executes the operation with incorrect parallelism, which results either in not using the full capacity because we have too few threads, or in a slow performance because we have too many threads.

We resolve this issue by setting the number of threads of each multi-threaded operator of a federated request according to the worker's level of parallelism. The worker inspects whether the operator inside an incoming request for executing an instruction is a multi-threaded operator to substitute the coordinator's thread size with its own parallelism. The applied operation-level-concurrency of the worker is set either individually via the configuration or to the default value, which corresponds to the number of available CPU cores. Therefore, the worker now executes the operations with its preferred level of parallelism.

Combined with the multi-threaded request processing of the previous section, the total thread count of the worker is determined hierarchically by the number of

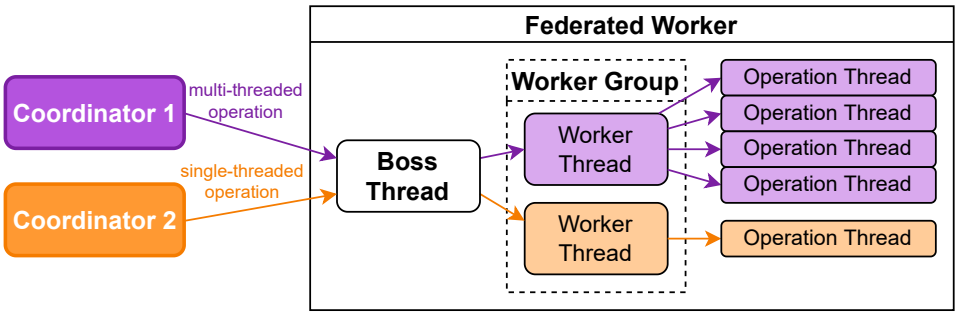

Figure 3.3: Multi-threaded Federated Worker

coordinators and the operation-level-parallelism (illustrated in Figure 3.3). Although this number can reach from 1 to the squared number of CPU cores, we expect it to scale at a reasonable level for conventional usage, since rarely the maximum number of concurrently operating coordinators—bound to the number of threads in Netty's worker group—executes operations at the same time. In addition, some operators can only be executed single-threaded and there might be blocking I/O operations involved which allow the execution of other threads. Furthermore, under heavy usage of the worker, we are more likely to over-provision CPU threads, which is generally a good idea in order to overlap fine-grained stalls, waits, and front-end-bound operations.

3.4 Federated Read Cache

Since the coordinators are now completely isolated at the federated worker, we identify a high probability that some tasks will be performed redundantly by different coordinators. Given that a federated worker usually refers to a certain dataset, coordinators addressing the worker often perform computations on the same data. To be able to address the data at the federated site, each coordinator first requests the worker to read the respective data. The worker then loads the data from the file system into the coordinator's individual execution context. Since reading from the file system is very slow, we recognize the advantage of caching this data for other coordinators to avoid redundant reads and redundant memory consumption in subsequent requests, which also impacts garbage collection overheads.

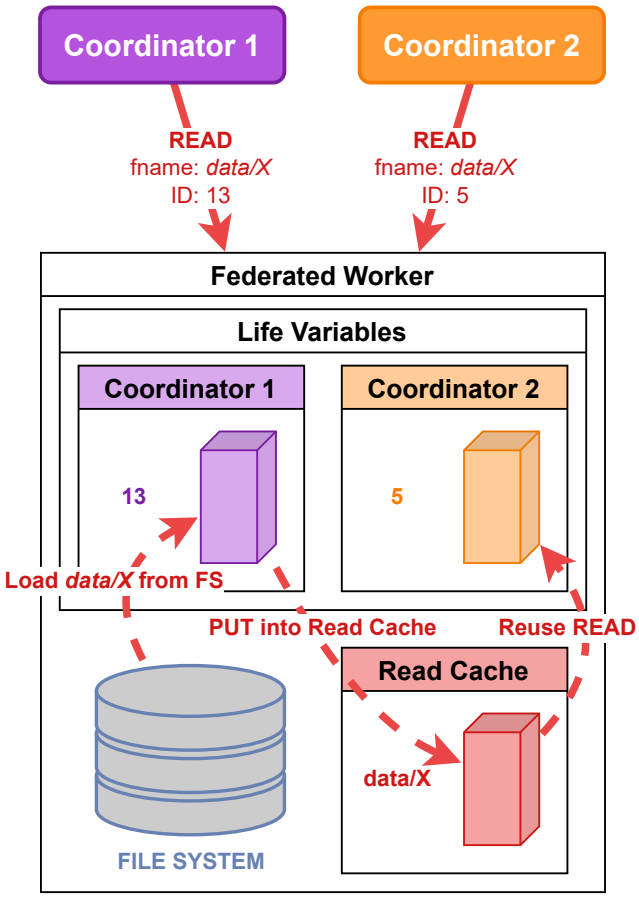

Figure 3.4: READ Reuse

We introduce the Federated Read Cache as a caching mechanism for read data objects at the federated worker. The Federated Read Cache is basically a map that associates a filename with a stored data object. Whenever the worker receives a request to read the data from a filename, it will first lookup the Federated Read Cache if the data object associated with that filename can be obtained from the cache. If the requested data is not present in the cache, however, the worker has to create a new data object from the filename, but directly puts a placeholder into the Federated Read Cache so that it is cached for the next request that comes along.

Figure 3.4 shows the scenario of two coordinators requesting to read the same data file. Since the request from Coordinator 1 is processed first and the data object is

not cached at this point, the worker has to load the data file from the file system and places it directly in the cache. Simultaneously, when processing the request from Coordinator 2, the cache contains either the data object or a placeholder to indicate that it is currently being read. Therefore, the worker waits, if necessary, until the data object is in the cache, and then fetches it from the cache by moving the pointer of the existing data object into the execution context of Coordinator 2 instead of loading the data again from the file system.

In SystemDS, a data object like a matrix or frame consists of the data block that contains the actual data and some meta information like the ID and the filename. Data blocks are kept in-memory as long as possible, meaning that even though the data is currently not in use, it will stay in memory until we have to evict it because of memory requirements. This eviction to the disk is realized by writing the data block to a buffer pool that might write it to the file system and ensure being able to reconstruct the object, and subsequently creating a soft reference to the corresponding in-memory block. The garbage collector then clears the entries of the soft references when memory is needed, requiring us to load the data block from the disk when it is needed again.

The automatic caching of data objects in SystemDS allows us to continuously populate the Federated Read Cache without ever removing one of its entries. Since the Federated Read Cache does not store the actual data but the corresponding data object, the contained data block is automatically written to the file system and evicted from memory when necessary. Therefore, the increase in memory consumption by the cache involves only the stored data objects (i.e., the meta-information) without the actual data.

3.5 Lineage-based Read Reuse

SystemDS comes with the integrated LIMA framework for fine-grained lineage tracing and reuse [45]. This framework maintains the lineage trace for all life variables in order to exactly identify an intermediate result. These lineage traces are then leveraged by the lineage-based reuse infrastructure, which allows for full and partial reuse of intermediates based on a dedicated reuse cache with cost-based evictions. However, the lineage cache is only built and utilized if the reuse by

lineage is enabled, which is activated by setting a dedicated flag that is passed to the program at startup.

As we already maintain a reuse cache if lineage-based reuse is activated, we also leverage it for read caching (see Section 3.4). Therefore, when reuse by lineage is enabled and the federated worker receives a request to read a data file, we first attempt to obtain it from the lineage cache instead of the Federated Read Cache and store the loaded data into the lineage cache after a cache miss. However, as the lineage cache is designed to store matrices, we still rely on the Federated Read Cache when reading frames, even in the case of the enabled lineage-based reuse.

Since the lineage-based reuse mechanism identifies the intermediates by their lineage trace, we need to associate the newly populated data with its lineage. Thus, we create a lineage item for each data file that is read from the file system, containing solely the filename, to construct a one-level lineage trace. Additionally, because the lineage cache eviction is cost-based, we measure the time needed for loading the data from the disk and tag the newly created cache entry accordingly.

The lineage cache is limited to a preconfigured absolute memory budget (default: 5% of the heap size). To ensure that this limit is not exceeded, the cache eviction constantly has to remove cache entries once the limit is reached. Therefore, the LIMA framework [45] introduced the lineage cache eviction with three configurable cost-based eviction policies (LRU, DAG-Height, Cost & Size), that take various statistics into account, such as the time for producing the cached object, the size of the object, and the usage of the cache entry.

By caching the read data in the lineage cache, we build a collective cache for all the cacheable intermediates. In contrast to separated caches, a joint cache has the advantage to avoid under-utilizing individual caching budgets. Therefore, the cache limit can be handled much better, since the cache entries are evicted according to their benefits, regardless of whether they are coming from reads or from intermediate results.

3.6 Lineage Trace Transfer

The LIMA framework [45], which is integrated into SystemDS [10], maintains the lineage for life variables at runtime in terms of their lineage trace of operations. A lineage trace is basically a directed acyclic graph (DAG) whose nodes contain information about the performed operations and whose edges represent data dependencies. Therefore, we are able to exactly identify a variable by its lineage trace.

Anyhow, the federated worker cannot infer the lineage of variables transmitted from the coordinator (broadcast variables), as it does not know what operations were performed by the coordinator to generate these variables. In order to maintain a complete set of lineage traces at the federated worker, we have to inform the worker about the lineage of a broadcast variable. Therefore, we serialize the entire lineage trace of a single variable before broadcasting and append the serialized lineage trace to the respective request. Then, after transmitting the request to the federated worker, we deserialize the lineage trace and relate it again to the corresponding broadcast variable.

The lineage trace of a variable consists of the information about the underlying operation and the lineage traces of the respective input variables of the operation in a recursive way. The nodes of this lineage DAG are called lineage items. Every input variable in the lineage trace represents a variable itself, and hence, comprises an individual lineage item with a unique ID.

Serialization: To serialize the lineage trace of a single variable, we traverse the DAG of lineage items in a depth-first manner. Starting from the lineage item of the respective variable, we append the string representation of the current lineage item to the overall serialization, which consists of the ID, the operation-code, and the IDs of the child nodes.

Deserialization: The deserialization is then realized exactly the other way around than the serialization. Traversing the serialized lineage items in the reverse direction allows for the recreation of the entire lineage DAG. Since during the previous serialization input nodes are processed after the referencing lineage item, they are processed before the referencing lineage item when deserializing, which allows to directly link the nodes of the lineage DAG to their children.

3.7 Federated Lineage Reuse

Due to the data-dependent applicability of pre-processing pipelines, the pre-processing steps are often identical for the same data, regardless of the following training pipeline. Hence, individual coordinators conducting different training pipelines on the same federated data usually still share the same pre-processing steps. Therefore, we recognize the high computational redundancy between the coordinators' isolated processing contexts at the federated worker. Additionally, iteratively performed ML pipelines show partial overlaps across the iterations because the iterative changes are relatively small (e.g., adapting the learning rate in gradient-based optimization), so that some subexpressions remain the same.

The LIMA framework [45] addresses redundant computations by introducing the lineage-based reuse cache for reusing intermediates. The lineage cache associates each cached intermediate with its respective lineage item, allowing for exact identification of the variable based on the corresponding lineage trace. Before computing a fresh intermediate, the lineage item for the computation is created in order to probe the cache for a match. If a match is found, the intermediate can be obtained from the cache instead of computing it again. The lineage trace comparison is possible by creating the lineage DAGs and comparing every node for equivalence.

With the Lineage Trace Transfer introduced in Section 3.6, we are now able to utilize the full potential of the lineage-based reuse mechanism at the federated worker. In order to eliminate computational redundancy between operations of different coordinators, we have to reuse intermediates across different processing contexts. Therefore, we build a single lineage cache at the worker, which jointly collects intermediates from all processing contexts. Although each context maintains its own lineage map, which associates variables with their corresponding lineage item, it is possible to compare the lineage traces from different lineage maps, since all individual operations from the entire trace are compared against each other. This capability allows for reusing computations between processing contexts using the same lineage cache.

Figure 3.5 shows the setup of two coordinators both performing the same computation at the federated worker. As the execution request from Coordinator 1 arrives first, the worker has to execute the requested computation and create the intermediate. However, before starting the computation, a placeholder-entry for this respective intermediate is placed in the cache to indicate that the value is

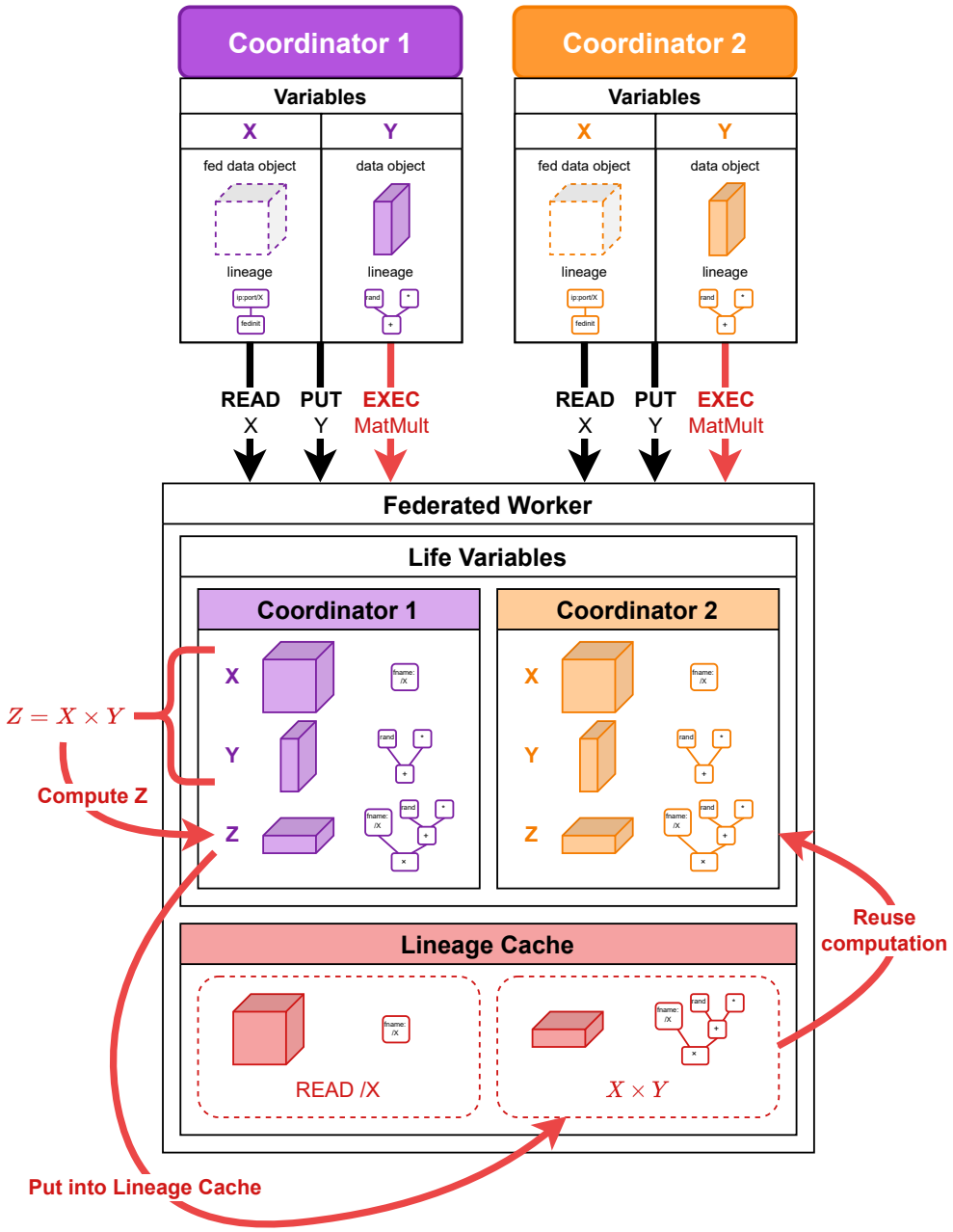

Figure 3.5: Lineage-based Reuse

currently being computed. This placeholder gets then immediately replaced with the intermediate value itself once the computation is completed. When requesting the same execution for Coordinator 2, we first probe the lineage cache for a match and obtain the corresponding cache entry. If the entry is a placeholder, the Coordinator 2 waits until the value of the placeholder is available and then reuses the intermediate. Otherwise, if the cache entry is not a placeholder, we can directly reuse the intermediate from the cache.

3.8 Response Serialization Reuse

Data pre-processing forms a crucial component in ML workflows and is often considered the central part of ML projects. In federated ML applications which only require the data pre-processing pipeline to be performed federated, the pre-processed data is sometimes pulled to the coordinator, where it is then analyzed locally to avoid the performance costs of the network communication. Before the resulting pre-processed data can be transmitted to the coordinator, it first needs to be serialized. Even though pre-processing pipelines often involve some sort of dimensionality reduction, these results can be very large. Therefore, the process of serializing the responses from the federated worker is a significant contributor to the overall execution time.

As mentioned in Section 3.7, pre-processing pipelines are strongly data-dependent. Hence, we expect coordinators to perform the same pre-processing steps for the identical data on a federated site. In a scenario where the coordinator pulls the pre-processed results from the worker to itself, it is very likely that the next coordinator which obtains the pre-processed results will fetch exactly the same data. In such a case, we are able to reuse the computation through the Federated Lineage Reuse mechanism (introduced in Section 3.7), but we have to serialize the reused intermediate for every transmission. Therefore, we see the possibility to also reuse serializations of transmitted results in order to save the time of redundantly serializing responses at the worker.

In SystemDS, the Netty framework is used to handle the network communication. Since Netty transmits a serialized network message as a ByteBuf, representing a sequence of bytes, we extend the lineage-based reuse cache with the ability to also store byte arrays in addition to matrices. Therefore, when we reply to the

coordinator with some data, we can now obtain the serialized byte array from the created buffer and put it into the lineage cache for the next time we need it.

To identify the newly inserted cache entry, we have to associate the byte array with a corresponding lineage item. As we cannot make use of the lineage from the underlying data object because it is already in use for the non-serialized intermediate, we indicate the serialization by attaching a lineage item for the serialization procedure to the original lineage. Hence, the lineage item for a serialized response is represented by a lineage item with the operation-code ”serialize” and a single input which is the lineage of the underlying data object. This additional lineage item allows to distinguish between the serialized response of a data object and the data object itself while keeping the lineage trace comparison as described in Section 3.7.

4 Experiments

Our experiments study the benefit and behavior of the introduced features and optimizations of MuTeFL by performing an ablation study of the individual techniques. We first conduct micro benchmarks to investigate the impact—in terms of performance and memory consumption—of each technique individually. Subsequently, we carry out algorithm-level experiments to examine the efficiency of the introduced reuse techniques. Finally, we study the overall effect of our MuTeFL system by simultaneously running several hyper-parameter optimization routines.

4.1 Experimental Setup

Hardware: In our experiments, we distinguish between two different nodes:

1. *α-node*: An α-node is equipped with two Intel Xeon Gold 6238R CPUs at 2.2-2.5 GHz having 28 physical cores and 56 virtual cores each. Each α-node has 768 GB DDR4 RAM at 2.933 GHz balanced across 6 memory channels per socket.
2. *β-node*: A β-node is equipped with a single AMD EPYC 7302 CPU at 3.0-3.3 GHz having 16 physical cores and 32 virtual cores. Each node has 128 GB DDR4 RAM at 2.933 GHz balanced across 8 memory channels.

Software: The software stack comprises Ubuntu 20.04.1 as operating system, OpenJDK Java 11.0.13, and SystemDS 3.0.0++. For our micro benchmarks and for the algorithm-level experiments, we use consistent JVM configurations of `-Xmx64g -Xms64g -Xmn6400m` for the federated workers, whereas the coordinators use the JVM configuration of `-Xmx32g -Xms32g -Xmn3200m`, except for the experiments of the Multi-threaded Event Loop where they use `-Xmx16g -Xms16g -Xmn1600m`. For the final performance experiment, we use the JVM configuration of

`-Xmx110g -Xms110g -Xmn11000m` for the federated worker and `-Xmx32g -Xms32g -Xmn3200m` for each coordinator.

Setup: Throughout our experiments, we distinguish between three different setups of the federated infrastructure:

1. *Local*: All coordinators and workers run on the same server node and communicate over the loopback interface with each other.
2. *LAN*: The coordinators all run on one server node and each federated worker runs on a separate node. Since all nodes belong to the same cluster, they connect to each other in a local area network (LAN) of two racks, connected via an HPE FlexFabric5710 48XGT switch.
3. *WAN*: In addition to LAN, we emulate a wide area network (WAN) by adding a NetEm [4] queuing discipline to the network interface. Inspired by the WAN setting of the experiments in the ExDRa paper [9], we thereby add a delay of 47.5 ±12.5 ms and limit the bandwidth to 2 MB/s. The exact command to limit the outgoing traffic is `tc qdisc add dev eth0 root netem delay 47.5ms 12.5ms 25% rate 2mbps`, executed on both the coordinators and the federated workers.

Datasets: We conduct our experiments on randomly generated synthetic data, varying the number of rows and keeping the number of columns constant. Furthermore, we perform the micro benchmarks with 500 columns and increase them to 1000 columns for the algorithm-level tests and the hyper-parameter optimizations.

Workloads: The tested workloads include various primitives for the micro benchmarks, linear regression (LM) and principal component analysis (PCA) followed by DBSCAN for the algorithm-level tests, and linear regression (LM) with and without PCA preprocessing for the overall performance experiment.

4.2 Micro Benchmarks

Multi-threaded Event Loop

We observe the impact of the Multi-threaded Event Loop (Section 3.3) on the system performance by comparing it to the same system with a Single-threaded Event Loop. For this experiment, we set up a WAN infrastructure with two β-nodes—one for the coordinators and the other for a single federated worker. We transfer a randomly generated matrix with 500 rows and 500 columns to the federated worker and read the federated matrix on the worker. Then, we perform a matrix multiplication of the federated matrix with the transferred matrix, followed by the sum, and pull the resulting scalar to the coordinator. We repeat this routine several times with both systems, varying the number of rows of the federated matrix and the number of simultaneously executing coordinators. Finally, we report the end-to-end times for the completion of all coordinators and show them in comparison between the two systems in Figure 4.1, where we observe a steadily increasing speedup as the number of concurrently executing coordinators increases.

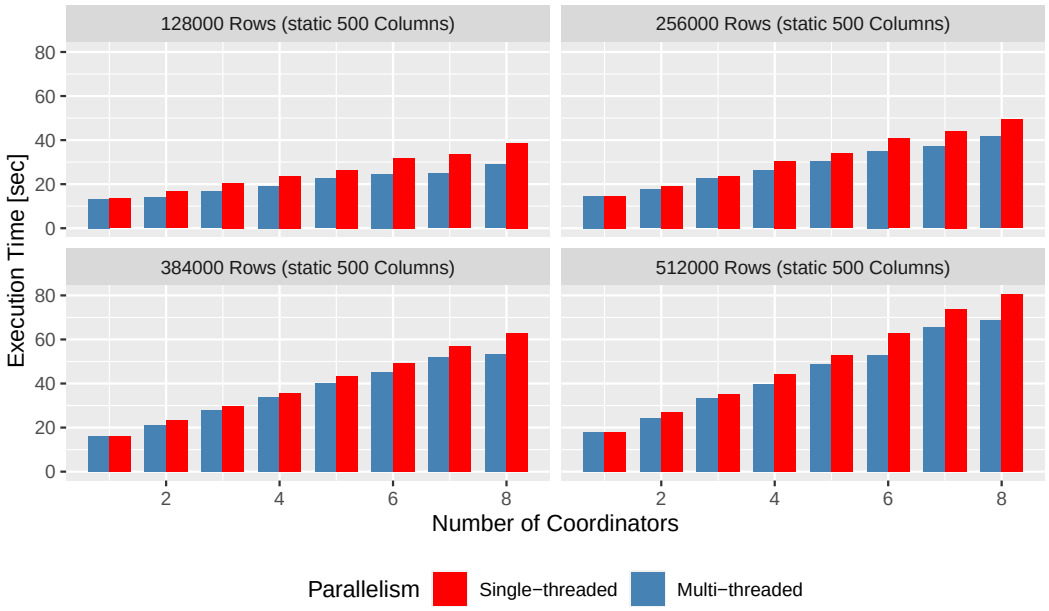

Figure 4.1: Multi-threaded Event Loop

Multi-threaded Operator

The dynamic adaptation of the operator parallelism at the worker (Multi-threaded Operator in Section 3.3) is useful in a setup with nodes of different parallelism. Therefore, we deploy a LAN federated infrastructure with an α-node for the federated worker and a β-node for the coordinator. This setup is intended to represent, for example, a data scientist running the coordinator on a laptop (low degree of parallelism) in order to analyze federated data from a worker deployed on a server (high degree of parallelism). We then perform the mean-operation, the matrix multiplication with a generated 500×500 matrix, and the sum-operation of the federated matrix, while varying the number of rows. Finally, we show the execution times of the mean-operation (Figure 4.2), the matrix multiplication (Figure 4.3), and the sum-operation (Figure 4.4) for the system with adapted parallelism and the reference system with static parallelism.

By adapting the parallelism on the worker, we now observe an improvement of approximately 2.25× for the mean-operation (shown in Figure 4.2), and an improvement of around 1.5× for the matrix multiplication (shown in Figure 4.3) as well as for the sum-operation (Figure 4.4).

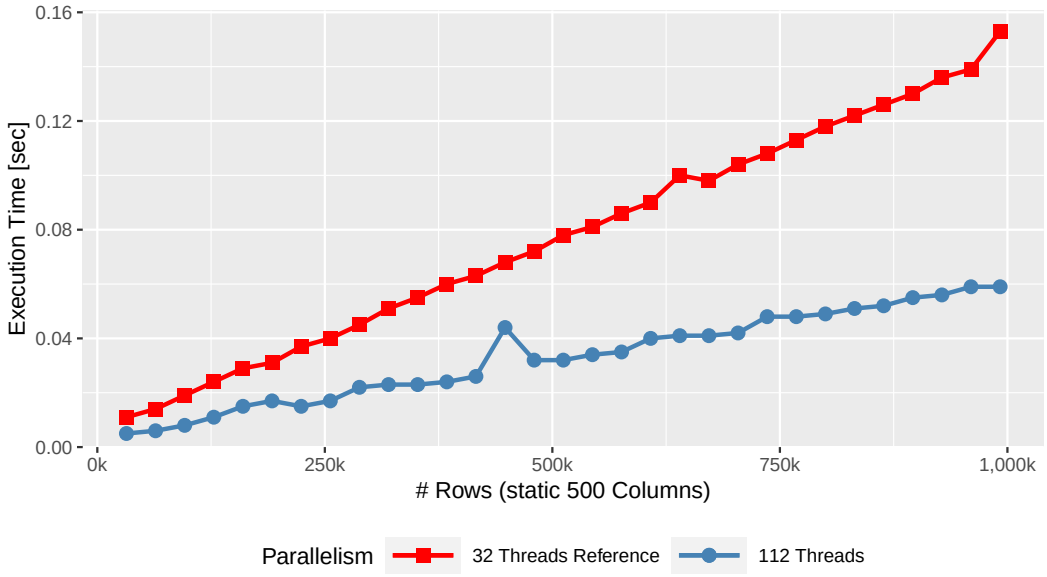

Figure 4.2: Multi-threaded Operator - Mean

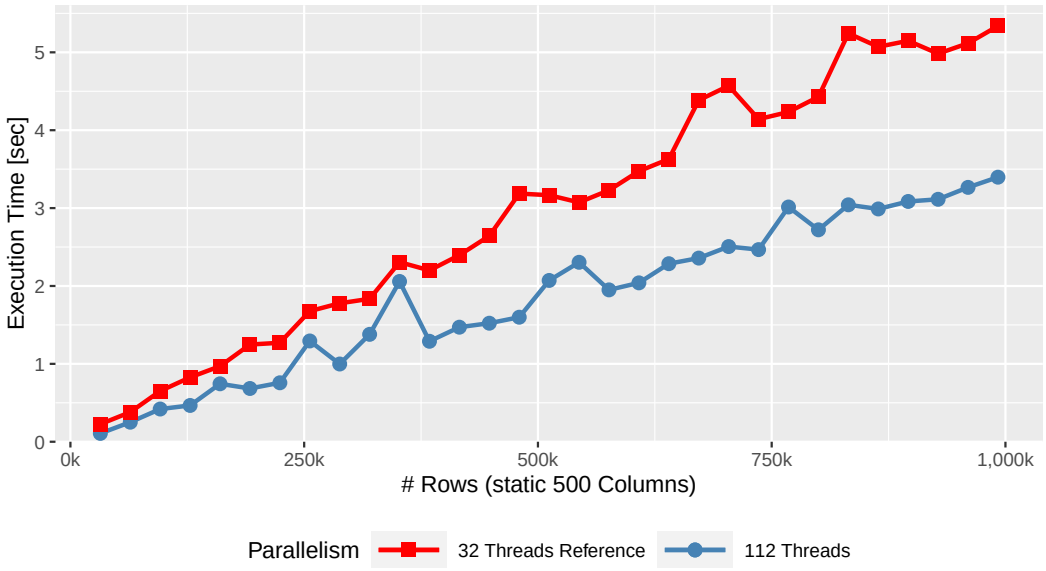

Figure 4.3: Multi-threaded Operator - Matrix Multiplication

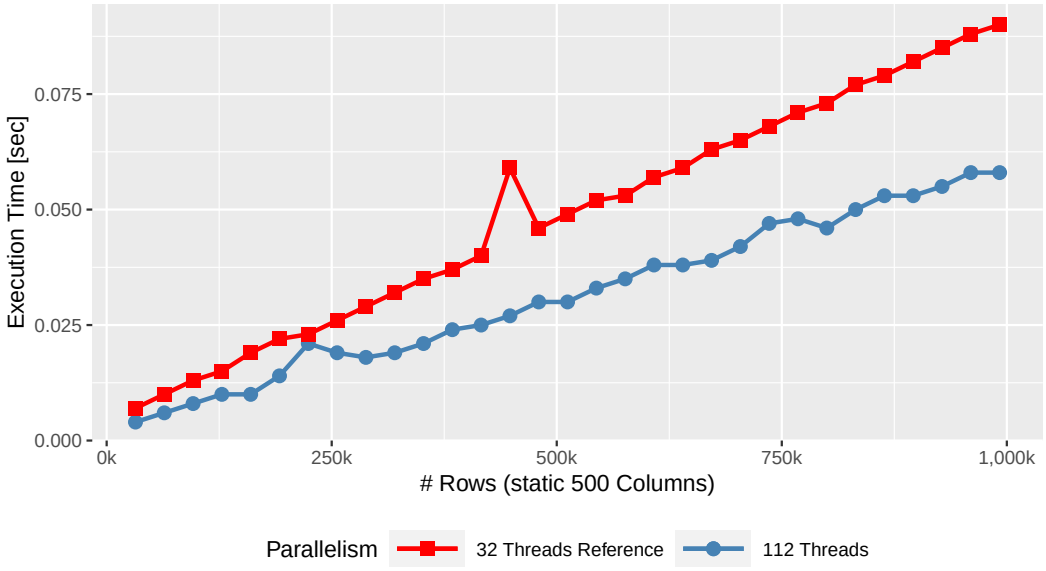

Figure 4.4: Multi-threaded Operator - Sum

Netty Buffer Allocation

During our work, we introduced the dynamic allocation of Netty's buffer for network transmissions. Instead of statically allocating the buffer size (256 Bytes by default) and increasing it as needed, we now estimate the size of the object to be transferred and directly allocate the estimated size, which results in a significant performance improvement for data transfers. To capture this improvement, we conduct experiments on the system before dynamic buffer allocation was introduced (commit 2087445[1]) and compare it to the system with dynamic buffer allocation (commit ba32d04[2]). In order to measure exact transmission times across the coordinator and the worker, we construct a Local federated infrastructure on a β-node with one coordinator and one federated worker, and perform an element-wise sum of the federated matrix with a broadcast matrix of the same size, while varying the number of rows. Subsequently, we pull 90% of the resulting matrix to the coordinator. In Figure 4.5, we present the transmission times for the broadcast request and for the response retrieval of both systems, which show an improvement of 2× for the dynamic buffer allocation compared to the static buffer allocation.

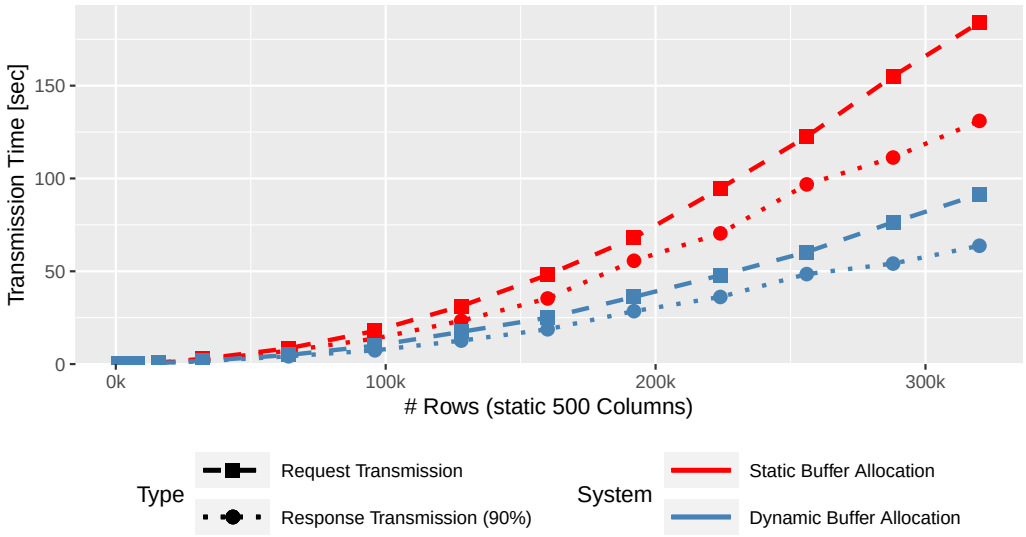

Figure 4.5: Request/Response Encoding

[1] https://github.com/apache/systemds/commit/2087445754b3352e13fe42724fe2884cc21ac0f7
[2] https://github.com/apache/systemds/commit/ba32d0499857f244406b44889a9ffff791e9cf92

Federated Read Reuse

The introduced reuse of read objects at the federated worker (Sections 3.4 and 3.5) eliminates redundant reads from the file system, and hence benefits both performance and memory consumption. Therefore, we evaluate the runtime for processing read requests and the respective memory consumption in three different settings: (1) without read reuse, (2) with the Federated Read Cache (Section 3.4), and (3) with Lineage-based Read Reuse (Section 3.5). For this experiment, we deploy a LAN federated infrastructure with one federated worker on a β-node and two simultaneously executing coordinators on a single β-node, both performing a sum-operation on the same federated data. We then measure and report the aggregated times for processing the read requests at the worker in Figure 4.6, the maximal memory consumption of the worker during execution in Figure 4.7, and the remaining memory consumption at the worker after execution and cleanup of both coordinators in Figure 4.8.

In Figure 4.6, we observe a slight speedup from the reuse systems compared to the reference system. Figure 4.7 clearly shows the overhead in memory consumption from the reference system caused by the duplicated read data objects that are avoided from the Federated Read Cache and the Lineage-based Read Reuse by reusing the in-memory data objects. Figure 4.8 shows the remaining memory consumption from the reuse systems that grows linearly with the data size and comes from the cached data objects, whereas the reference system consumes almost no memory after the execution.

Since, when the coordinators are executed concurrently, the times for processing the read requests contain the waiting time for the other coordinator to complete the read from the file system and put the data object into the cache, we additionally perform the same experiment with sequentially started coordinators and report the respective times from the reads in Figure 4.9. For that reason, the reuse systems need only half as much time as the reference system to process the read requests (shown in Figure 4.9), because the first coordinator performs the actual read from the file system and the second can directly take the data object from the cache instead of reading it from the file system again.

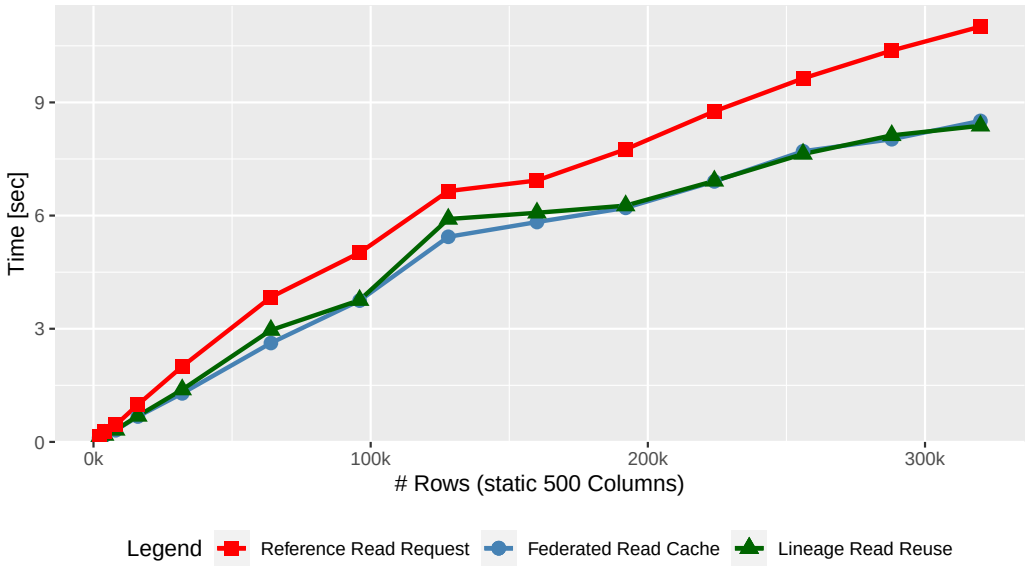

Figure 4.6: Read Reuse Performance - Concurrent Coordinator Execution

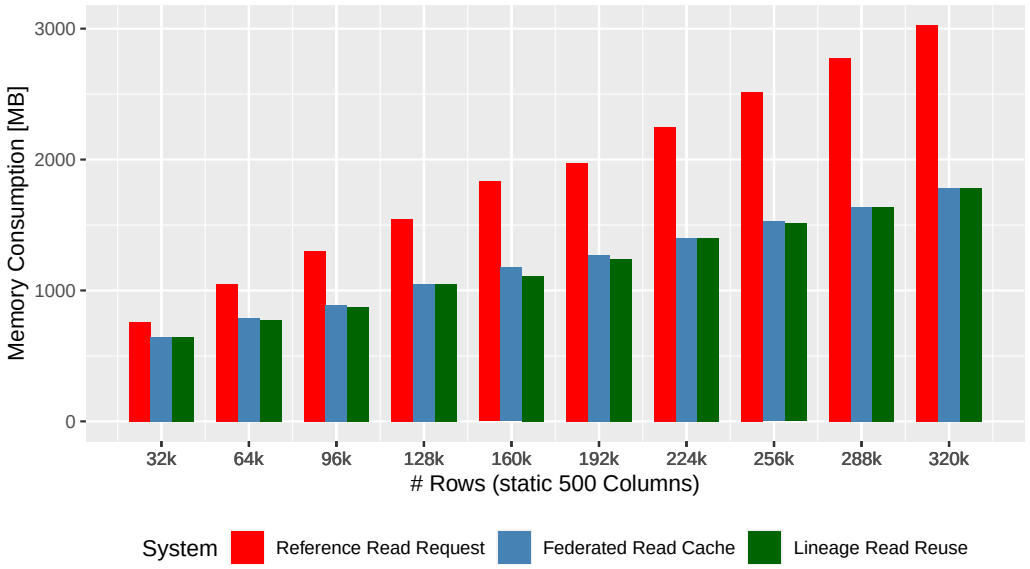

Figure 4.7: Read Reuse Memory Consumption

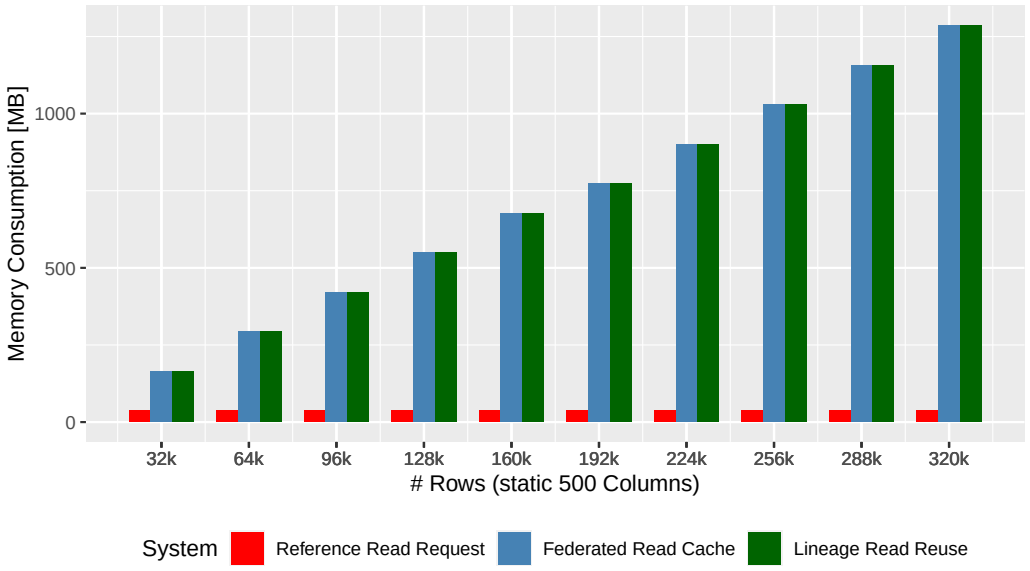

Figure 4.8: Read Reuse Remaining Memory Consumption

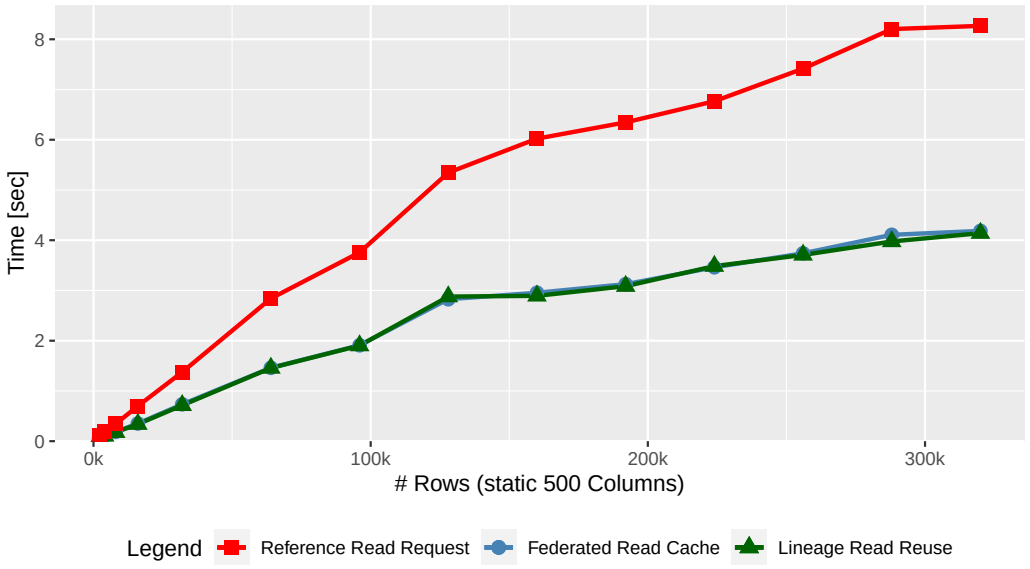

Figure 4.9: Read Reuse Performance - Sequential Coordinator Execution

Lineage Trace Transfer

Since the introduced Lineage Trace Transfer (Section 3.6) is an addition to the base system, we evaluate the overhead incurred by the additional serialization and deserialization of the lineage trace, as well as the overhead in the transmission of the larger federated request. For this purpose, we deploy a WAN infrastructure with one coordinator on a β-node and a single federated worker on a β-node. For the experiment, we then generate a 1000×500 matrix on the coordinator and another on the federated worker. In order to evaluate different sizes of the lineage trace of the matrix on the coordinator, we modify the matrix in a for loop by performing a simple plus operation with a scalar which adds one lineage item per iteration. In this way, we vary the number of lineage items in our transferred lineage trace and measure the time required to serialize and deserialize the lineage trace (Figure 4.11). Additionally, we measure the time overhead from the broadcast transmission (joint time from request and response transmission) by subtracting the execution time of the instruction at the worker from the time at the coordinator and report the resulting transmission times in Figure 4.10.

The overhead in the transmission time caused by including the lineage trace is clearly visible in Figure 4.10, as we can see, for example, in the experiment with 50k lineage items, where we have an overhead of almost 0.8 seconds compared to the transmission without lineage trace. In addition, Figure 4.11 shows the overhead from serializing and deserializing the lineage trace, which adds another overhead of 0.64 seconds for the case of 50k lineage items. Therefore, the transfer of the lineage trace with 50k lineage items adds a total overhead of 1.44 seconds in our experiments, corresponding to an overhead of 0.0288 milliseconds per lineage item.

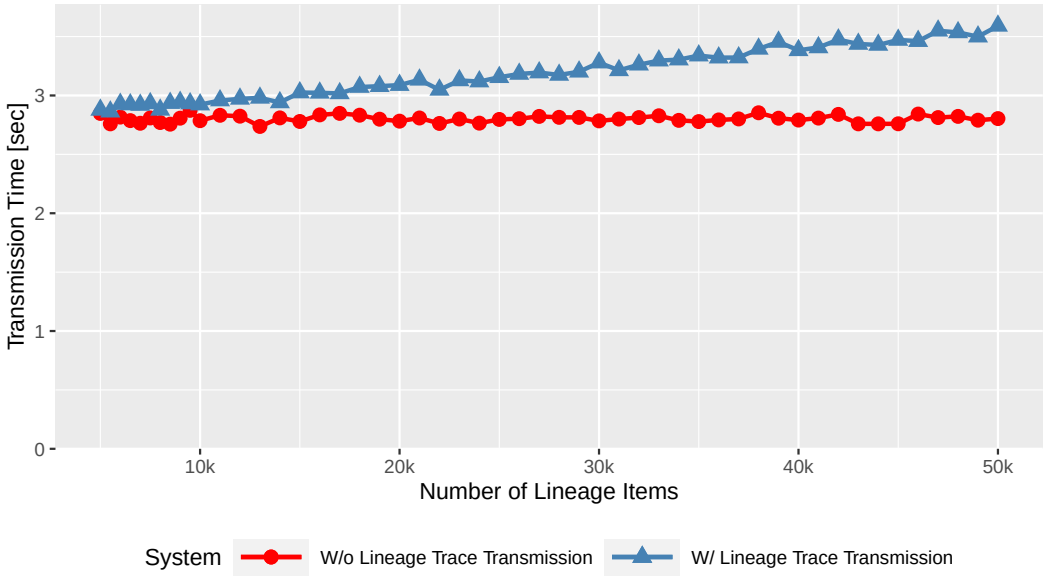

Figure 4.10: Lineage Trace Transmission

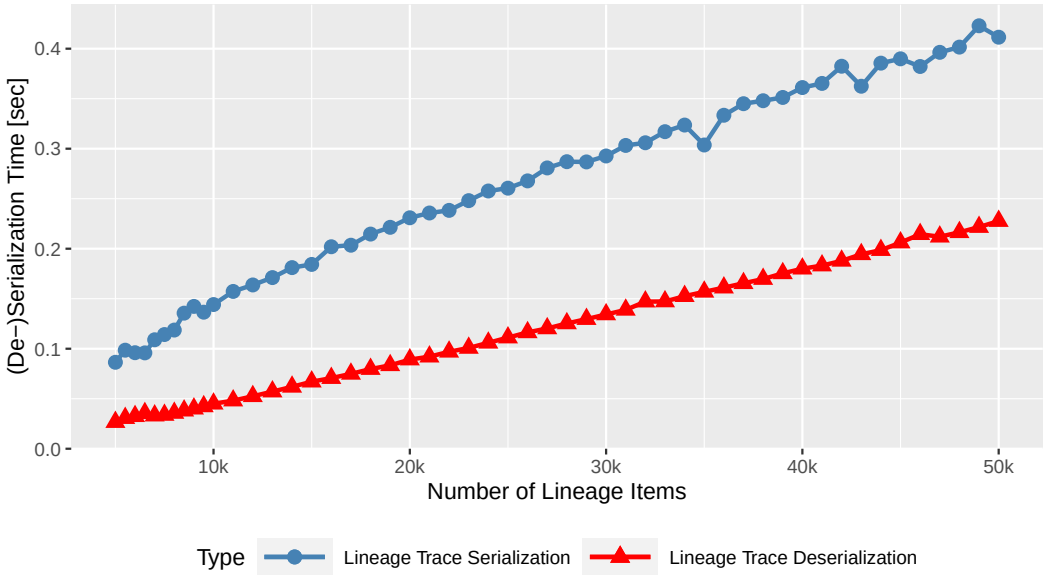

Figure 4.11: Lineage Trace Serialization and Deserialization

4.3 Algorithm Performance

In the Algorithm Performance experiments, we observe the impact of the introduced techniques of Federated Lineage Reuse (Section 3.7) and Response Serialization Reuse (Section 3.8) on the level of a simple algorithm execution.

Lineage Reuse at the Federated Worker

Since the Federated Lineage Reuse (Section 3.7) saves computation time but also adds a certain overhead in runtime due to the reuse checks and caching, we are interested in whether and how much time is actually saved by this reuse. Therefore, we set up a Local federated infrastructure on a β-node for multiple coordinators and a single federated worker. With each coordinator, we train a linear regression model on a federated matrix with 1000 features and a federated vector, while varying the number of sequentially executed coordinators and the number of rows of both federated data objects. During execution, we capture the time spent on reuse (checking the reuse cache and putting the data into the cache) and the time saved by reusing at the federated worker, and compare these times in Figure 4.12.

From the time stacks in Figure 4.12, we can directly recognize that the saved time outweighs the time spent in the case of multiple coordinators. However, when performing the experiment with one coordinator, we cannot reuse any intermediate since there are no redundant federated computations in a single execution. Figure 4.12 also shows the increase in the saved computation time as the number of rows increases, while the time for the reuse check remains the same since it does not depend on the data size.

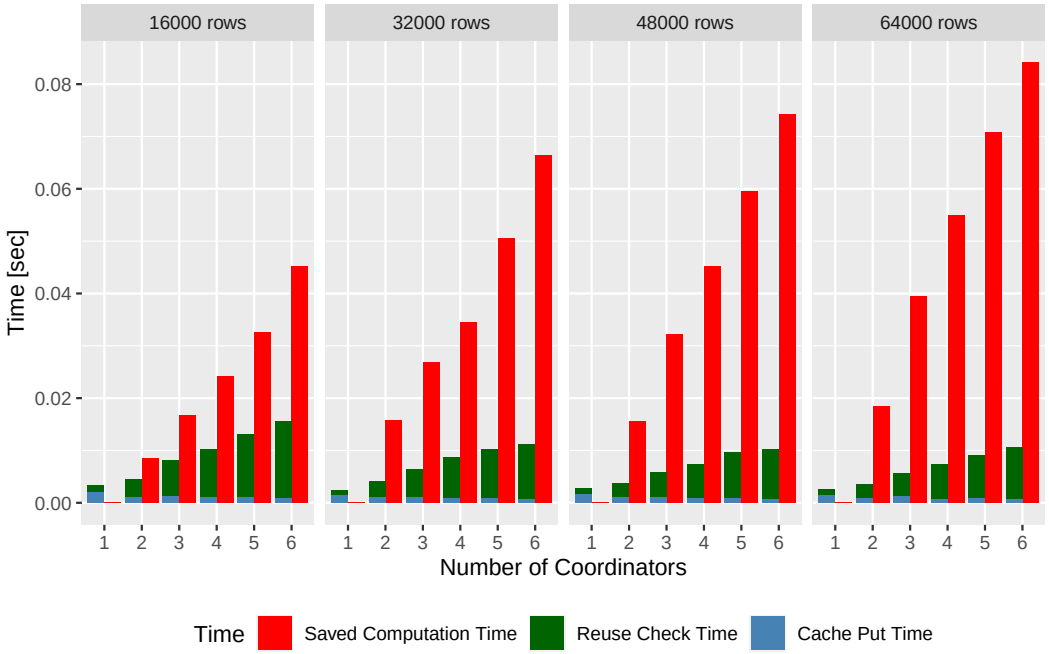

Figure 4.12: Lineage-based Reuse Times

Response Serialization Reuse

Similar to the Federated Lineage Reuse, Response Serialization Reuse saves serialization time but also consumes runtime to probe the cache and put the serialization into the cache. Therefore, we are again interested in whether and how much time is saved by reusing the serializations. In this experiment, we deploy a Local federated infrastructure on an α-node for multiple coordinators and with two federated workers. We create a row-partitioned federated matrix of 1000 columns with equal-sized partitions on the federated workers. Subsequently, we perform PCA without dimensionality reduction (keeping all components of the data) as a preprocessing step before clustering the data using DBSCAN, while varying the number of sequentially executed coordinators and the number of rows. Throughout this experiment, we record the time saved by reusing the serializations and the time spent on checking the reuse cache and putting the serialized bytes into the cache, and compare these for each worker in Figure 4.13.

Similar to the previous experiment, Figure 4.13 shows no saved time for the executions with one coordinator. With the increasing number of coordinators, we can see a slight advantage of the saved time for the experiment with 8000 rows and with 16000 rows, whereas this advantage is clearly visible for the experiment with 24000 rows. In addition, we observe an increase in time for the reuse check, coming from writing the reused serialized bytes into the buffer for the network transmission.

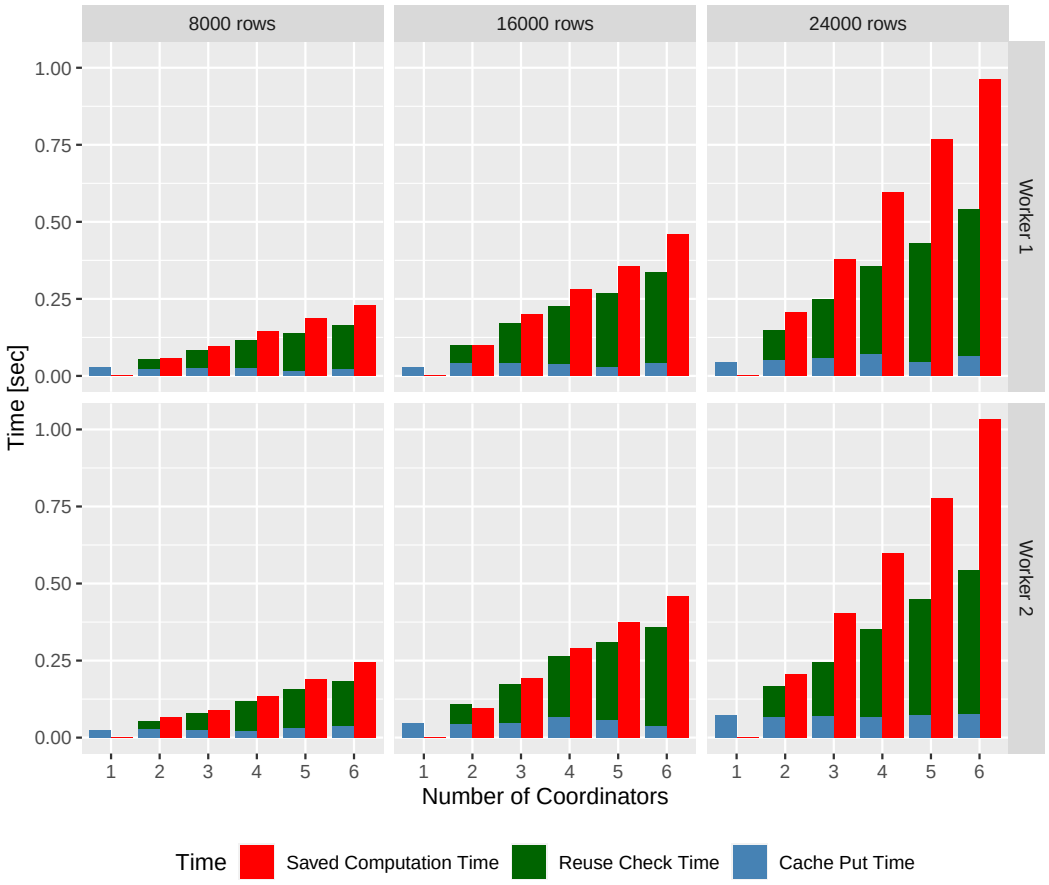

Figure 4.13: Response Serialization Reuse Times

4.4 Hyper-parameter Tuning Performance

In this experiment, we try to simulate four data scientists who concurrently perform hyper-parameter optimization of a linear regression model. Two data scientists perform PCA on the data as preprocessing step and also include the selected number of components in the hyper-parameters to optimize, whereas the other two data scientists train the model on the plain data. Table 4.1 shows the tested hyper-parameters for each data scientist (i.e., coordinator).

Table 4.1: Overview of Hyper-parameters

Coordinator	λ	intercept	tolerance	k (PCA)
I	$[10^{-9}, 10^{-2}]$	$[0, 6]$	$[10^{-12}, 10^{-6}]$	N/A
II	$[10^{-3}, 10^{0}]$	$[0, 2]$	$[10^{-12}, 10^{-9}]$	$\{300, 400, 500\}$
III	$[10^{-7}, 10^{0}]$	$[-3, 3]$	$[10^{-10}, 10^{-4}]$	N/A
IV	$[10^{-5}, 10^{-2}]$	$[0, 2]$	$[10^{-10}, 10^{-7}]$	$\{250, 500, 750\}$

We deploy a LAN federated infrastructure with one β-node for the four coordinators and one β-node for the federated worker. The coordinators iterate over all combinations of the tested hyper-parameters, in every iteration training the model with the respective hyper-parameters and computing the adjusted R^2 score to find the best parameters. In the experiment, we start all four coordinators concurrently with a shared federated matrix of 4000 rows and 1000 columns.

We conduct this experiment for three different baselines:

1. *Reference System*: In order to compare to a reference system, we disable the Federated Read Cache, the parallelization level adaptation from the Multi-threaded Operator, and the Lineage Trace Transfer. Additionally, we change the event loop to be single-threaded and deactivate the lineage-based reuse on both the coordinators and the federated worker.
2. *MuTeFL-W System*: We include all concepts introduced in this work, but deactivate the lineage-based reuse on the coordinators while activating it for the federated worker.
3. *MuTeFL-L System*: Includes all concepts of this work, with lineage-based reuse activated on both the coordinators and the federated worker.

We report the total execution times for each individual coordinator of the three baselines in Figure 4.14, where we observe a great difference to the MuTeFL-L

System coming from local eliminations of federated computations. However, we also see the benefit of the MuTeFL-W system, as the four coordinators clearly show faster execution times. Additionally, we measure the maximal memory consumption on the federated worker during execution (shown in Figure 4.15), where we recognize a small overhead from both MuTeFL systems compared to the Reference System.

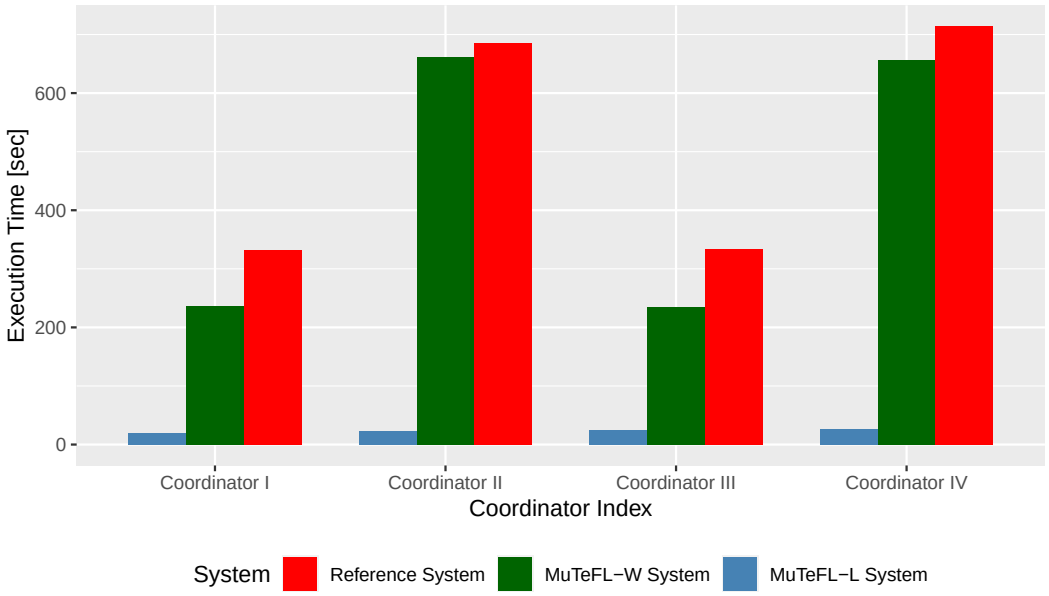

Figure 4.14: Hyper-parameter Optimization - Coordinator Execution Times

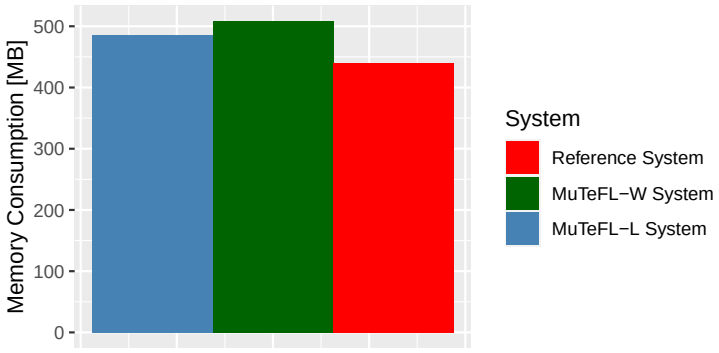

Figure 4.15: Hyper-parameter Optimization - Worker Memory Consumption

In addition to the total execution times, we are interested in the times of Federated Lineage Reuse, Lineage-based Read Reuse, and Response Serialization Reuse. Therefore, we record the saved time, the time to perform the reuse checks, and the time to put the values into the cache, and present them distinctly for the three optimizations in Figure 4.16 for MuTeFL-W and in Figure 4.17 for MuTeFL-L. We can conclude from both figures that all three reuse techniques positively affect the runtime in our experiments. The difference in reuse times between the MuTeFL-W system (Figure 4.16) and the MuTeFL-L system (Figure 4.17) again comes from the elimination of federated computations from the MuTeFL-L system. Note that the saved computation time does not necessarily correspond to the difference in total execution times from Figure 4.14, since it is measured during the first computation of each intermediate and summed up as the intermediate is reused.

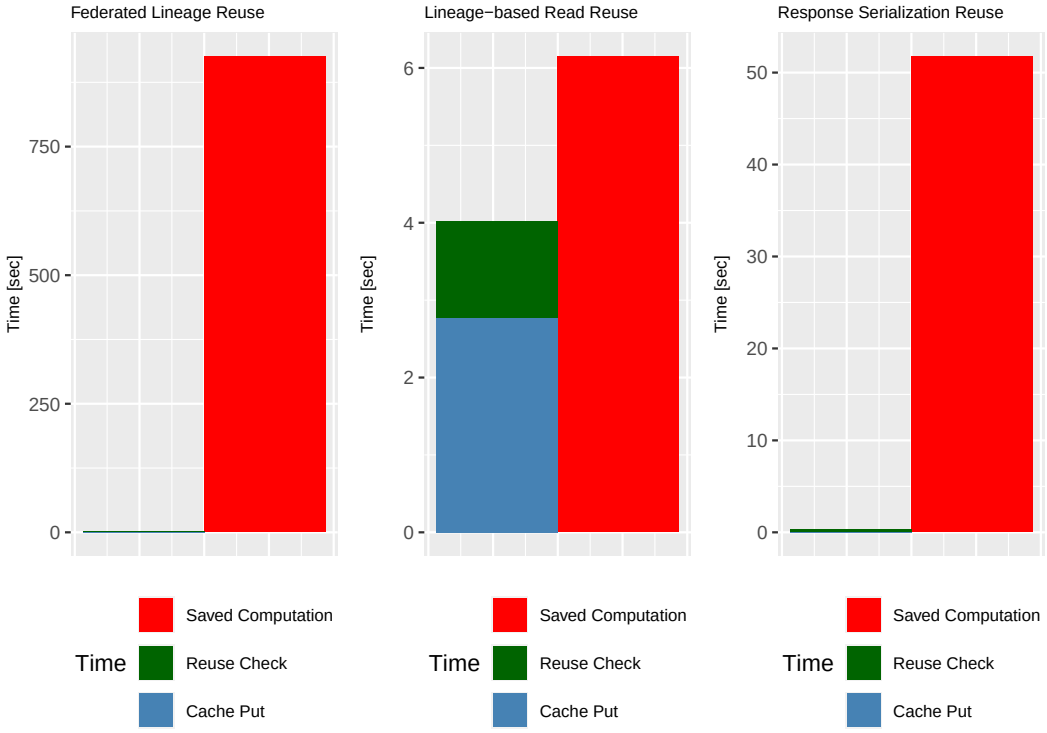

Figure 4.16: Hyper-parameter Optimization - MuTeFL-W Reuse Times

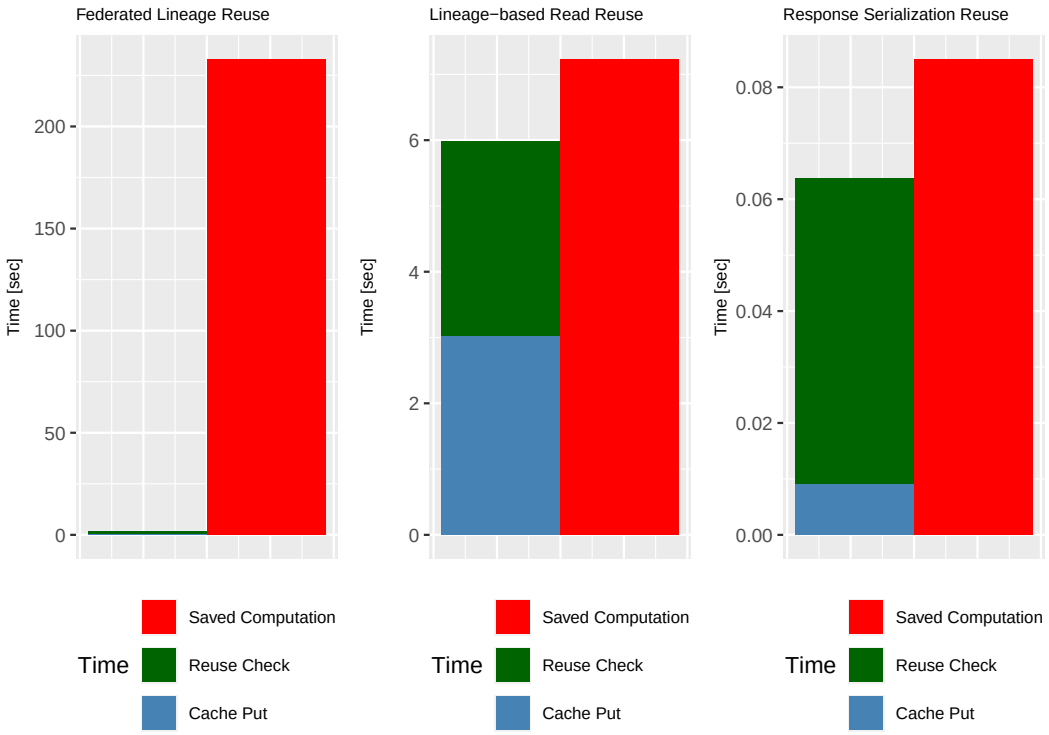

Figure 4.17: Hyper-parameter Optimization - MuTeFL-L Reuse Times

Since the introduced concept of Lineage Trace Transfer creates an overhead in runtime, we count the number of lineage trace transfers and measure the times for serializing and deserializing the lineage traces. Table 4.2 reports the respective statistics that do not indicate a substantial impact on the total execution times from Figure 4.14.

Table 4.2: Lineage Trace Transfer - (De-)Serialization Times

	# Transfers	Serialization	Deserialization
MuTeFL-W	5056	0.7874 sec	1.8708 sec
MuTeFL-L	2191	0.5906 sec	0.8914 sec

5 Related Work

The MuTeFL concept extends the privacy-enhancing technology of Federated Learning by combining techniques from various research areas such as Multi-tenant Cloud DBMS and Reuse of Intermediates in ML systems. Since MuTeFL relies on Federated Learning, it is highly related to the fields of Federated Databases, Federated Query Processing, and Privacy-preserving ML.

Federated Learning: First mentioned in 2016 [39], the term Federated Learning (or Collaborative Learning) denotes a privacy-preserving ML technology that aims to train ML models on distributed data without consolidating and revealing the sensitive data to a central party. Early applications of Federated Learning (FL) focused on training models directly on user devices [15, 27, 48]. The great success of FL resulted in the formation of further applications in industry as well as in health care [58, 46, 13, 42, 59]. Over the years, research on Federated Learning has been conducted in many different directions and thus, experienced a number of developments, such as performance and accuracy improvements [53, 31, 57], the privacy enhancement of the algorithms [16], robustness of the FL system [36], and the heterogeneity of federated sites [17].

Privacy-preserving ML: Federated Learning emerges as an important technique in the field of privacy-preserving ML. Besides Federated Learning, active research is done on the techniques of Differential Privacy (DP) [20, 21], Homomorphic Encryption (HE) [7, 30, 34], and Multi-party Computation (MPC) [40, 33]. In contrast to FL, HE and DP require a central consolidation of the data. Although in MPC the data remains distributed, it suffers from a high communication overhead in big data workflows coming from the exchange of secrets between the parties. Recent work on privacy-preserving ML also considers the combination of FL with HE to protect against threats from the server when the model itself contains sensitive data or raw data can be reconstructed from the aggregates (e.g., batch size of 1) [37].

Multi-tenant Cloud DBMS: The multi-tenant utilization of distributed resources and the objective of tenant isolation are important principles of both MuTeFL and Multi-tenant Cloud DBMS [43]. Additionally, also the caching of intermediates is a present topic in the field of Multi-tenant Cloud DBMS [44]. Examples of such systems are Microsoft SQL Azure [3] and Apache Cassandra [1].

Reuse of Intermediates: The optimization of the MuTeFL system strongly relies on the lineage-based reuse technique of the LIMA-framework [45]. This approach of identifying redundant computations and reusing them is related to the Automatic Materialization of KeystoneML [54] and the Common Subexpression Elimination of TensorFlow [6], with the difference that LIMA is a runtime technique with compiler-assistance. Due to the conditional control flow, common subexpression elimination cannot eliminate certain types of redundancy. The technique of eliminating redundancy of operations is also related to the suppression of partial redundancies in the field of compilers [41, 19].

Federated Database System: The main component of a Federated Database System (FDBS) is the Software to manipulate distributed data across autonomous databases—the Federated Database Management System (FDBMS) [29, 52, 8]. The task of an FDBMS to manage data that is distributed among multiple databases is of a similar nature to the job of a federated infrastructure with data at the federated sites. Recent work on FDBMS even deals with the distribution of data among different types of databases (graph-based, document-based, and relational databases). Examples are wrappers in Db2 [5], CloudMdsQL [32], BigDAWG [25], Myria [56], and Apache Drill [2, 28]. In the context of Federated Database Systems, Federated Query Processing (FQP) techniques provide a unified interface to access the distributed data [22]. FQP engines have the responsibility to translate a federated query expressed in the global schema to a local query which can be executed at the individual sites. This concept of translation is similar to the Federated Runtime Plan creation of SystemDS' federated backend (Section 2.1.4). Example FQP engines are Ontario [24], FedX [51] based on SparQL Federated Query [14, 47], and MULDER [23].

6 Conclusions

To summarize, we introduced MuTeFL, an extension of a federated backend to support multi-tenant execution. The introduced technique of Tenant Distinction together with the Federated Lookup Table ensure robust and safe management of multiple coordinators at the federated worker. The Multi-threaded Event Loop and the Multi-threaded Operator allow for efficient utilization of the resources at the federated worker. To eliminate I/O time from redundant file system reads, we maintain the Federated Read Cache or leverage the lineage cache if activated by the means of Lineage-based Read Reuse to reuse already read data objects. With the Lineage Trace Transfer, we complete the lineage map at the federated worker, including the lineage from broadcast data objects, which we then leverage by enabling lineage-based reuse at the federated worker (within and across isolated coordinators) with the Federated Lineage Reuse. Response Serialization Reuse further extends the lineage cache by supporting serialized bytes from Federated Responses and reusing them if the same response is sent multiple times to save the time from redundantly serializing the content of the response.

In conclusion, as the field of federated learning expands rashly, MuTeFL offers the possibility for deployment with long-running, server-like federated workers and for applications with multiple coordinators. Our conducted experiments show the feasibility as well as the robustness of a MuTeFL system. Furthermore, we demonstrated in the experiments that the introduced optimizations are useful. However, there still exist many opportunities for further development and optimization of MuTeFL, for instance an extension of the broadcast communication to avoid unnecessary transmissions, or a dynamic adjustment of the used parallelism on the worker depending on its overall workload.

Bibliography

[1] Apache Cassandra. https://cassandra.apache.org/. Accessed: 2022-07-07.

[2] Apache Drill. https://drill.apache.org/. Accessed: 2022-07-07.

[3] MS Azure SQL. https://azure.microsoft.com/en-us/products/azure-sql/. Accessed: 2022-07-07.

[4] netem. https://wiki.linuxfoundation.org/networking/netem. Accessed: 2022-08-13.

[5] Wrappers and wrapper modules. https://www.ibm.com/docs/en/db2/11.5?topic=systems-wrappers-wrapper-modules. Accessed: 2022-08-16.

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2016.

[7] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, 51(4), jul 2018.

[8] L. G. Azevedo, E. F. S. Soares, R. Souza, and M. F. Moreno. Modern federated database systems: An overview. In *ICEIS*, 2020.

[9] S. Baunsgaard, M. Boehm, A. Chaudhary, B. Derakhshan, S. Geißelsöder, P. M. Grulich, M. Hildebrand, K. Innerebner, V. Markl, C. Neubauer, S. Osterburg, O. Ovcharenko, S. Redyuk, T. Rieger, A. Rezaei Mahdiraji, S. B. Wrede, and S. Zeuch. *ExDRa: Exploratory Data Science on Federated Raw Data*, page 2450–2463. Association for Computing Machinery, New York, NY, USA, 2021.

[10] M. Boehm, I. Antonov, S. Baunsgaard, M. Dokter, R. Ginthoer, K. Innerebner, F. Klezin, S. Lindstaedt, A. Phani, B. Rath, B. Reinwald, S. Siddiqi, and S. Wrede. Systemds: A declarative machine learning system for the end-to-end data science lifecycle. 2020. 10th Conference on Innovative Data Systems Research, CIDR 2020 ; Conference date: 12-01-2020 Through 15-01-2020.

[11] 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, and S. Tatikonda. Systemml: Declarative machine learning on spark. *Proc. VLDB Endow.*, 9(13):1425–1436, sep 2016.

[12] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, B. McMahan, T. Van Overveldt, D. Petrou, D. Ramage, and J. Roselander. *Towards Federated Learning at Scale: System Design.* MLSys, Palo Alto, CA, USA, 2019.

[13] T. S. Brisimi, R. Chen, T. Mela, A. Olshevsky, I. C. Paschalidis, and W. Shi. Federated learning of predictive models from federated electronic health records. *International Journal of Medical Informatics*, 112:59–67, 2018.

[14] C. Buil-Aranda, M. Arenas, and O. Corcho. Semantics and optimization of the sparql 1.1 federation extension. In G. Antoniou, M. Grobelnik, E. Simperl, B. Parsia, D. Plexousakis, P. De Leenheer, and J. Pan, editors, *The Semantic Web: Research and Applications*, pages 1–15, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[15] M. Chen, R. Mathews, T. Y. Ouyang, and F. Beaufays. Federated learning of out-of-vocabulary words. *ArXiv*, abs/1903.10635, 2019.

[16] K. Cheng, T. Fan, Y. Jin, Y. Liu, T. Chen, and Q. Yang. Secureboost: A lossless federated learning framework. *IEEE Intelligent Systems*, 36:87–98, 2021.

[17] E. Diao, J. Ding, and V. Tarokh. Heterofl: Computation and communication efficient federated learning for heterogeneous clients. *ArXiv*, abs/2010.01264, 2021.

[18] K. Dooley and I. Brown. *Cisco IOS Cookbook: Field-Tested Solutions to Cisco Router Problems.* Cookbooks (O'Reilly). O'Reilly Media, 2006.

[19] K.-H. Drechsler and M. P. Stadel. A solution to a problem with morel and renvoise's "global optimization by suppression of partial redundancies". *ACM Trans. Program. Lang. Syst.*, 10(4):635–640, oct 1988.

[20] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Third Conference on Theory of Cryptography*, TCC'06, page 265–284, Berlin, Heidelberg, 2006. Springer-Verlag.

[21] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3–4):211–407, aug 2014.

[22] K. M. Endris. Federated query processing over heterogeneous data sources in a semantic data lake. 2020.

[23] K. M. Endris, M. Galkin, I. Lytra, M. N. Mami, M.-E. Vidal, and S. Auer. *Querying Interlinked Data by Bridging RDF Molecule Templates*, pages 1–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2018.

[24] K. M. Endris, P. D. Rohde, M.-E. Vidal, and S. Auer. Ontario: Federated query processing against a semantic data lake. In *DEXA*, 2019.

[25] V. Gadepally, P. Chen, J. Duggan, A. Elmore, B. Haynes, J. Kepner, S. Madden, T. Mattson, and M. Stonebraker. The bigdawg polystore system and architecture. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2016.

[26] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[27] A. Hard, K. Rao, R. Mathews, S. Ramaswamy, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and D. Ramage. Federated learning for mobile keyboard prediction, 2018.

[28] M. Hausenblas and J. Nadeau. Apache drill: Interactive ad-hoc analysis at scale. *Big data*, 1 2:100–4, 2013.

[29] D. Heimbigner and D. McLeod. A federated architecture for information management. *ACM Trans. Inf. Syst.*, 3(3):253–278, jul 1985.

[30] E. Hesamifard, H. Takabi, M. Ghasemi, and C. Jones. Privacy-preserving machine learning in cloud. In *Proceedings of the 2017 on Cloud Computing Security Workshop*, CCSW '17, page 39–43, New York, NY, USA, 2017. Association for Computing Machinery.

[31] S. P. Karimireddy, S. Kale, M. Mohri, S. J. Reddi, S. U. Stich, and A. T. Suresh. Scaffold: Stochastic controlled averaging for federated learning. In *ICML*, 2020.

[32] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jiménez-Peris, R. Pau, and J. Pereira. Cloudmdsql: Querying heterogeneous cloud data stores with a common language. *Distrib. Parallel Databases*, 34(4):463–503, dec 2016.

[33] A. Lapets, F. Jansen, K. D. Albab, R. Issa, L. Qin, M. Varia, and A. Bestavros. Accessible privacy-preserving web-based data analysis for assessing and addressing economic inequalities. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*, COMPASS '18, New York, NY, USA, 2018. Association for Computing Machinery.

[34] J. Li, X. Kuang, S. Lin, X. Ma, and Y. Tang. Privacy preservation for machine learning training and classification based on homomorphic encryption schemes. *Information Sciences*, 526:166–179, 2020.

[35] L. Li, Y. Fan, M. Tse, and K.-Y. Lin. A review of applications in federated learning. *Computers Industrial Engineering*, 149:106854, 2020.

[36] T. Li, S. Hu, A. Beirami, and V. Smith. Ditto: Fair and robust federated learning through personalization. In *ICML*, 2021.

[37] A. Madi, O. Stan, A. Mayoue, A. Grivet-Sébert, C. Gouy-Pailler, and R. Sirdey. A secure federated learning framework using homomorphic encryption and verifiable computing. In *2021 Reconciling Data Analytics, Automation, Privacy, and Security: A Big Data Challenge (RDAAPS)*, pages 1–8, 2021.

[38] B. McMahan and D. Ramage. Federated learning: Collaborative machine learning without centralized training data. Google AI Blog, 2017. Accessed: 2022-06-22.

[39] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y. Arcas. Communication-efficient learning of deep networks from decentralized data. 2016.

[40] P. Mohassel and Y. Zhang. Secureml: A system for scalable privacy-preserving machine learning. *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38, 2017.

[41] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, feb 1979.

[42] N. I. Mowla, N. H. Tran, I. Doh, and K. Chae. Federated learning-based cognitive detection of jamming attack in flying ad-hoc network. *IEEE Access*, 8:4338–4350, 2020.

[43] V. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri. Sqlvm: Performance isolation in multi-tenant relational database-as-a-service. In *CIDR 2013*. 6th Biennial Conference on Innovative Data Systems Research, January 2013.

[44] V. R. Narasayya and S. Chaudhuri. Multi-tenant cloud data services: State-of-the-art, challenges and opportunities. *Proceedings of the 2022 International Conference on Management of Data*, 2022.

[45] A. Phani, B. Rath, and M. Boehm. Lima: Fine-grained lineage tracing and reuse in machine learning systems. In *LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems*, Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 1426–1439, 2021.

[46] Prayitno, C. R. Shyu, K. T. Putra, H.-C. Chen, Y.-Y. Tsai, K. S. M. T. Hossain, W. Jiang, and Z. Shae. A systematic review of federated learning in the healthcare area: From the perspective of data properties and applications. *Applied Sciences*, 2021.

[47] B. Quilitz and U. Leser. Querying distributed rdf data sources with sparql. In S. Bechhofer, M. Hauswirth, J. Hoffmann, and M. Koubarakis, editors, *The Semantic Web: Research and Applications*, pages 524–538, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[48] S. Ramaswamy, R. Mathews, K. Rao, and F. Beaufays. Federated learning for emoji prediction in a mobile keyboard, 2019.

[49] Y. M. Saputra, D. T. Hoang, D. N. Nguyen, E. Dutkiewicz, M. D. Mueck, and S. Srikanteswara. Energy demand prediction with federated learning for electric vehicle networks. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2019.

[50] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. 01 2000.

[51] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. Noy, and E. Blomqvist, editors, *The Semantic Web – ISWC 2011*, pages 601–616, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[52] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 22(3):183–236, sep 1990.

[53] J. So, B. Guler, and A. S. Avestimehr. Turbo-aggregate: Breaking the quadratic aggregation barrier in secure federated learning. *IEEE Journal on Selected Areas in Information Theory*, 2:479–489, 2021.

[54] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics, 2016.

[55] A. Vulimiri, C. Curino, B. Godfrey, K. Karanasos, and G. Varghese. Wanalytics: Analytics for a geo-distributed data-intensive world. In *CIDR*, 2015.

[56] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whitaker, and S. Xu. The myria big data management and analytics system and cloud services. In *CIDR*, 2017.

[57] W. Wu, L. He, W. Lin, R. Mao, C. Maple, and S. A. Jarvis. Safa: A semi-asynchronous protocol for fast federated learning with low overhead. *IEEE Transactions on Computers*, 70:655–668, 2021.

[58] J. Xu and F. Wang. Federated learning for healthcare informatics. *Journal of Healthcare Informatics Research*, 5:1 – 19, 2021.

[59] W. Yang, Y. Zhang, K. Ye, L. Li, and C.-Z. Xu. Ffd: A federated learning based method for credit card fraud detection. In K. Chen, S. Seshadri, and L.-J. Zhang, editors, *Big Data – BigData 2019*, pages 18–32, Cham, 2019. Springer International Publishing.