ABSTRACT
Federated learning allows training machine learning (ML) models without central consolidation of the raw data. Variants of such federated learning systems enable privacy-preserving ML, and address data ownership and/or sharing constraints. However, existing work mostly adopt data-parallel parameter-server architectures for mini-batch training, require manual construction of federated runtime plans, and largely ignore the broad variety of data preparation, ML algorithms, and model debugging. Over the last years, we extended Apache SystemDS by an additional federated runtime backend for federated linear-algebra programs, federated parameter servers, and federated data preparation. In this paper, we share the system-level compiler and runtime integration, new features such as multi-tenant federated learning, selected federated primitives, multi-key homomorphic encryption, and our monitoring infrastructure. Our demonstrator showcases how composite ML pipelines can be compiled into federated runtime plans with low overhead.

CCS CONCEPTS
• Information systems → Data management systems; • Computing methodologies → Machine learning.

KEYWORDS
Federated Learning, Federated Raw Data, Monitoring

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

1 INTRODUCTION
Data privacy requirements, data ownership, and other constraints on data sharing render central data consolidation—and training machine learning (ML) models on this data—infeasible in many applications. For this reason, privacy-preserving ML and data sharing have been addressed with a variety of techniques. Figure 1 shows the resulting spectrum of data sharing, covering techniques with different tradeoffs of privacy, performance, and utility for ML applications. Private data and heavy-weight techniques like fully homomorphic encryption (FHE) [2], multi-party computation (MPC) [21], and differential privacy [15] provide the strongest privacy guarantees with limited utility. Other techniques like anonymized or surrogate data [9] are more prone to revealing the raw data.

Federated learning: In contrast, federated learning [7, 16] overcomes sharing limitations by training models on the decentralized, federated data. Often data-parallel parameter servers [11, 18, 27] and similar distribution strategies [1, 19] are adopted [16]. A central coordinator initializes the model; federated devices or sites now act as workers, pull the current model, perform, for instance, a forward and backward pass through a neural network to compute gradients, and push these gradients or updated models to the coordinator, where the updates are accumulated and shared with all workers. Federated learning adds an interesting and practical design point to the spectrum of data sharing by sharing aggregates—that reveal data distributions but not the raw data—with moderate overhead.

Limitations of Existing Work: Existing work on federated learning made valuable contributions to system infrastructure [7], optimization algorithms [16, 23, 25], and the integration with privacy enhancing technologies [12, 13, 20, 29, 31]. However, existing work primarily focuses on data-parallel parameter servers (for mini-batch training), which ignores data preparation and feature transformations, a wide variety of batch ML algorithms, as well as end-to-end ML pipelines. Systems like TensorFlow federated [14]...
allow for general federated analytics, but require users or developers to construct federated runtime plans. Creating new libraries of federated ML algorithms and primitives \cite{12, 13, 29} becomes difficult for complex, composite ML pipelines.

**Federated Learning in SystemDS:** A key observation underpinning Apache SystemDS \cite{4} is that state-of-the-art data cleaning and debugging techniques are based on machine learning, and thus, can be expressed in linear algebra. For this reason, we build a hierarchy of built-in functions for data preparation, training, and debugging on top of a domain-specific language for ML training and scoring. An optimizing compiler then generates efficient hybrid runtime plans of local, in-memory and distributed operations. This design allows a seamless integration of federated learning by dedicated compilation and runtime techniques. In the larger ExDra project, SystemDS is already used as the backbone for federated learning \cite{3}. Key differentiators are generic abstractions for federated data and request types; federated instructions for a variety of linear algebra operations and statistical functions; federated parameter servers; federated data preparation; as well as the automatic generation of valid and efficient federated runtime plans.

**Contributions:** In this demonstration proposal, we share the end-to-end system integration of SystemDS’ federated backend, including new features on compiling federated plans, multi-tenant federated learning, and selected federated primitives (Section 2), as well as a monitoring tool for federated learning (Section 3). All features are fully integrated in Apache SystemDS\cite{1}. The demonstration scenarios (Section 4) then utilize these components and provide an in-depth understanding of how ML pipelines are compiled into federated runtime plans, and how these plans are executed.

## 2 SYSTEM ARCHITECTURE

In this section, we recap the system architecture of Apache SystemDS’ federated backend \cite{3} and describe recent extensions of handling federated data, compiling federated runtime plans, multi-tenant federated learning, and selected federated operations.

### 2.1 Federated Runtime Backend

**Federated Data:** Our federated backend comprises abstractions for federated data, federated linear algebra operations for ML algorithms, federated parameter servers for mini-batch training, and federated feature transformations \cite{3}. Federated frames or matrices are virtual objects whose physical parts are located at SystemDS workers in federated sites. Such federated matrices can be read from meta data, or constructed at script level:

```r
F = federated(
    addresses=list("node1:8001/Finput1.csv",
    "node2:8001/Finput2.csv"),
    ranges=list(list(0,0), list(50K,70), #50K rows
    list(50K,8), list(120K,70))); #70K rows
```

**F = scale(X=F, center=TRUE, scale=TRUE);**

User scripts at the coordinator (e.g., scale and k-Means above) are compiled into a hierarchy of program blocks and instructions, and all instructions on federated data become federated operations.

**Federated Requests:** Federated requests are implemented via a small set of federated requests (READ_VAR, PUT_VAR, GET_VAR, EXEC_INST, EXEC_UDF, and CLEAR). For example, a matrix-vector multiplication on row-partitioned federated data broadcasts the vector via PUT_VAR, executes partial matrix-vector multiplications via EXEC_INST, and optionally obtains and concatenates the federated outputs. Batches of these requests are executed concurrently via Netty remote procedure calls (RPCs) for all federated workers.

**Partitioning and Replication:** Federated data allows for arbitrary disjoint partitioning. In order to simplify federated operations, we employ specific partitioning and replication types (FType). In this context, a row-partitioned federated matrix has an FType. ROW (partitioning ROW, replication NONE), while a broadcast variable has FType. BROADCAST (partitioning NONE, replication FULL).

**Broadcasting:** Broadcasting in a federated setting is similar to broadcasting in data-parallel frameworks like Spark \cite{30} or Dask \cite{24}. However, meta data about federated ranges allows more control. We provide internal primitives for broadcast and broadcastSliced. For a row-partitioned matrix, a matrix-vector multiplication needs the full vector at every worker, whereas a vector-matrix multiplication only needs vector slices, avoiding unnecessary data transfer and memory consumption. Furthermore, in contrast to RDD and broadcast variables in Spark, all broadcasts are managed as federated data. Thus, a sliced broadcast simply becomes a federated matrix, and all federated operations apply.

### 2.2 Compiling Federated Plans

Besides a basic runtime conversion—which robustly works even for conditional control flow—we now support configurable types for compiling federated runtime plans. The compilation of federated plans facilitates debugging, controls the execution of cost-optimal plans, and adheres to user-provided privacy constraints.

- **None:** Obtain the federated data and run local operations.
- **Runtime:** Convert all operations on federated inputs to federated operations during runtime (Section 2.1) \cite{3}.
- **Compile-Fed_All:** Compile federated operations with the objective of keeping intermediates federated if possible.
- **Compile-Fed_Heuristic:** Compile federated operations with the same heuristics as Runtime (e.g., collect agg. vectors).
- **Compile-Fed_Cost-based:** Recursively enumerate optimal subplans for interesting properties of intermediates (e.g., FType.ROW, FType.COL, Local) and finally, select the global cost-optimal plan from terminal operators (write and print).

Table 1 shows the runtime of L2-SVM (support vector machines) on federated data, where Fed-All is—even in a local area network—almost 2x slower than Fed-Heuristic/Fed-Cost due to the RPC latency of many federated vector operations for a line search on the gradients in an inner loop. Instead, these operations can be performed at the coordinator because the gradients are aggregates.

**Table 1: L2SVM Training w/ Federated Planners (3 workers).**

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Fed-None</th>
<th>Fed-Runtime</th>
<th>Fed-All</th>
<th>Fed-Heuristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2s</td>
<td>5.9s</td>
<td>9.4s</td>
<td>5.8s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\footnote{The source code is available at https://github.com/apache/systemds, in packages runtime/controlprogram/federated, runtime/instructions/fed, and hops/fedplanner.}
specific FType of inputs. The federated planners then iterate over the program structure, and propagate the type of intermediates. Operations can be forced to federated operations, and their outputs can be fixed to local or federated. Additional primitives exist for collecting (fed-to-local) and broadcasting (local-to-fed) data.

**Privacy Constraints:** We further support propagating and checking privacy constraints—both during compilation and runtime—from the inputs through the program. Example constraints include types such as public, private-aggregate, and private partitions or features; and valid purposes such as specific operations. If existing, these constraints change the objective to a constrained minimization of cost subject to these privacy constraints, which raises errors if no plan satisfies the constraints. In the future, we will also integrate privacy enhancing technologies (e.g., homomorphic encryption) in this framework to handle strictly private data.

### 2.3 Multi-tenant Federated Learning

Federated workers are long-running server processes that receive concurrent requests from multiple coordinators. This scenario requires robust multi-tenant federated learning as shown in Figure 2.

**Tenant Isolation:** For robust isolation, we use a tree of tenant-specific execution contexts, where each context also holds a map of life variables (mapping from variable ID to unpinned data objects). Tenants are uniquely identified by a combination of their coordinator process ID (included in requests) and IP address as seen from the worker. For each request, we construct the coordinator ID, lookup the tenant context, and read/store intermediates in this context. This approach guarantees that no intermediates are overwritten, even if coordinators run on the same host. Furthermore, a single coordinator might run a parallel for-loop and spawn multiple concurrent requests. The parfor worker IDs are also included in the requests and create an additional hierarchy level of contexts.

**Event-Loop:** Workers listen for federated requests, execute these requests, and return federated responses with optional SSL-encryption. The configuration of this event-loop has high impact on performance. By default, we limit the number of concurrent connections to the number of virtual cores because typically the number of coordinators is moderate. For incoming EXEC_* requests, the workers further set the operation parallelism to the number of virtual cores for adapting to heterogeneous worker hardware and avoiding under-utilization with time-varying workloads.

**Lineage-based Reuse:** Fully isolating the individual tenants leads to unnecessary redundancy. First, the federated data is read and kept in memory multiple times. Second, exploratory data science exhibits high redundancy in and across ML pipelines of a single tenant, but also across multiple tenants. We address this challenge by a dedicated read cache and lineage-based reuse [22] with placeholders for synchronization during execution. Files are assumed unchanged during uptime of federated workers. We then trace lineage of all operations, which takes the lineage of inputs and constructs the output lineage. This lineage is the key in a process-wise lineage cache across all contexts. For obtaining the lineage of received data (in PUT_VAR requests), coordinators can send the data’s lineage, and if unavailable, we construct a high-probability identifier based on CRC checksums and additional meta data.

### 2.4 Selected Federated Primitives

**Basic Federated Operations:** Meanwhile SystemDS reached feature completeness for all operations that can be supported on federated data. Basic operations that map cleanly to federated operations include feature transformations (encode, apply, decode), a variety of linear algebra and arithmetic operations, statistical functions, aggregations, indexing, and reorganizations. Recently, we also added fused operator pipelines [6], and second-order operations like map(F, v -> v.substring(5)). With these federated operations, many DSL-based built-in functions for data preparation, ML algorithms, and model debugging can process federated data.

**Advanced Federated Operations:** Several operations require multi-pass implementations. These operations include cumulative aggregates, and remove empty row/column operations, which have cross-row-/column dependencies but can be efficiently computed via aggregation trees [5]. Other operations like ordering are impossible on row-partitioned data. Quantiles—which in turn rely on ordering—are, however, the basis for many statistics, data cleaning primitives (e.g., outlier removal), and equi-height histograms. Accordingly, we support federated quantiles via a recursive histogram refinement approach. First, we determine the min and max values of a row-partitioned feature and split this range into 256 buckets. Second, every federated worker counts the frequency of values per bucket on its local data. Third, the coordinator collects and aggregates these histograms. On the cumulative sum of these bucket frequencies, we determine in which bucket the required quantile falls (e.g., 0.5 quantile for the median). This bucket is then again split into 256 buckets and we repeat steps 2 and 3 until we get the result. Due to the large fan-out, federated quantiles only require few round-trips and the transferred data is very small.

**Federated Parameter Server:** For mini-batch training, SystemDS provides a dedicated paramserv built-in function for data-parallel (multi-threaded or distributed) parameter-servers. This parameter server infrastructure has been extended for federated operations by respecting the boundaries of federated data, and handling data imbalance (different data size) and skew (different data distribution). With parameters for update frequencies, this infrastructure is able to run FedAvg and similar optimization algorithms [16, 23]. Recently, we added support for multi-key homomorphic encryption [20] of gradients, which relies on Microsoft’s SEAL library [8, 26], and guards against plain-text exchange of gradients.

### 3 MONITORING INFRASTRUCTURE

As an additional debugging and demonstration tool, we are currently building a dedicated monitoring infrastructure for Apache SystemDS Federated. In this section, we describe the backend and frontend (see Figure 3) designs of this infrastructure in detail.
This demonstration will utilize Apache SystemDS, its federated backend of Apache SystemDS for compiling and executing federated end-to-end ML pipelines. New components include the compilation of federated runtime plans, multi-tenant federated learning with robust isolation and reuse, new federated primitives for data preparation and debugging, as well as a new monitoring infrastructure. In contrast to existing systems, federated learning in SystemDS seamlessly applies to a wide variety of DSL-based ML algorithms and other primitives, reuses existing runtime infrastructure, and thus makes federated learning practical.

ACKNOWLEDGEMENTS
This work is a result of the ExDrRa project, funded through the bilateral program “ICT of the Future – Smart Data Economy” by the Austrian Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK, 873838). Contributions by Sebastian B. Wrede were made in the DDAI COMET module (funded by BMK, BMDW, FFG, SFG, and industrial partners).