Databricks Lakeguard: Supporting Fine-grained Access Control and Multi-user Capabilities for Apache Spark Workloads

Martin Grund
Stefania Leone
Herman van Hövell
Sven Wagner-Boysen
Sebastian Hillig
sigmod25-lakeguard@databricks.com
Databricks
San Francisco, CA, USA

Hyukjin Kwon
David Lewis
Jakob Mund
Polo-Francois Poli
Lionel Montrieux
sigmod25-lakeguard@databricks.com
Databricks
San Francisco, CA, USA

Othon Crelier
Xiao Li
Reynold Xin
Matei Zaharia
Michalis Petropoulos
Thanos Papathanasiou
sigmod25-lakeguard@databricks.com
Databricks
San Francisco, CA, USA

Abstract

Enterprises want to apply fine-grained access control policies to manage increasingly complex data governance requirements. These rich policies should be uniformly applied across all their workloads. In this paper, we present Databricks Lakeguard, our implementation of a unified governance system that enforces fine-grained data access policies, row-level filters, and column masks across all of an enterprise's data and AI workloads. Lakeguard builds upon two main components: First, it uses Spark Connect, a JDBC-like execution protocol, to separate the client application from the server and ensure version compatibility. Second, it leverages container isolation in Databricks' cluster manager to securely isolate user code from the core Spark engine. With Lakeguard, a user's permissions are enforced for any workload and in any supported language, SQL, Python, Scala, and R on multi-user compute. This work overcomes fragmented governance solutions, where fine-grained access control could only be enforced for SQL workloads, while big data processing with frameworks such as Apache Spark relied on coarse-grained governance at the file level with cluster-bound data access.

CCS Concepts

• **Computer systems organization** → **Distributed architectures**; **Client-server architectures**; **Cloud computing**.

Keywords

Apache Spark, Data Governance, Unity Catalog, Query Processing, User Isolation, Security, Lakehouse

ACM Reference Format:

Martin Grund, Stefania Leone, Herman van Hövell, Sven Wagner-Boysen, Sebastian Hillig, Hyukjin Kwon, David Lewis, Jakob Mund, Polo-Francois Poli, Lionel Montrieux, Othon Crelier, Xiao Li, Reynold Xin, Matei Zaharia, Michalis Petropoulos, and Thanos Papathanasiou. 2025. Databricks Lakeguard: Supporting Fine-grained Access Control and Multi-user Capabilities for Apache Spark Workloads. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3722212.3724433

SIGMOD-Companion '25, June 22–27, 2025, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3724433

1 Introduction

Enterprises need support for fine-grained access control policies to manage increasingly complex data governance requirements. For example, common use cases include limiting the visibility of data between different departments, such as finance and HR, filtering out PII (personally identifiable information), or granting users time-constrained access to data. SQL data warehouses have rich support for defining fine-grained access control (FGAC), including features such as dynamic views, row-level filters, and column masks, as well as more dynamic, attribute-based access control [2, 27]. In contrast, big data processing frameworks such as Apache Spark have different and often more limited ways to manage access controls. To reduce complexity and ensure data governance is always enforced, enterprises need a unified way to specify and enforce rich access control policies efficiently across all their data and AI workloads.

Today, it is common for data and AI workloads to access the same data, commonly stored in inexpensive cloud storage [30] using open file formats such as Delta [9] and Iceberg [6] following the Lakehouse [35] architecture. Data governance is defined on top, based on a centralized data catalog, such as Unity Catalog [29], Glue [3], or Hive [28]. Enforcement of the governance rules is defined by the implementation of individual engines and therefore fragmented. Previously, big data processing services relied on one of the following approaches to enforce governance: limit the user surface to declarative non-programmable interfaces [33], only support coarse-grained table-level access controls [6], column-level encryption in Parquet [21], expensive external filtering [4], or by sacrificing utilization [19]. To achieve unification, a central governance layer is required, covering big data processing, data warehousing, and AI, independently of the data use case. In Databricks, we unified all of the above requirements in one data platform, which allows

customers to seamlessly manage and govern not only data but assets such as models, feature functions and pipelines, all part of the value chain for deriving insights from data.

In this paper, we present Databricks Lakeguard, a system to enforce rich data governance rules uniformly and efficiently across data and AI workloads. From traditional SQL-based data warehousing workloads to workloads in data engineering and machine learning in Python or Scala/Java. Lakeguard supports complex access control policies familiar to SQL users, including dynamic views, row-level filters, and column-level masks. Lakeguard consists of two main components. First, we contributed Spark Connect to the Apache Spark project, a query execution protocol in the spirit of JDBC that allows the running of Spark applications locally or remotely [26]. Second, Lakeguard implements user code isolation using containers on the Databricks cluster manager to securely run user code in Spark applications. The combination of both components allows users to use the full power of Spark while Databricks enforces rich access control policies, uniformly and efficiently, across all languages and workload types. Lakeguard can also serve applications that need full access to the underlying machine (e.g., GPU ML workloads) via external fine-grained access control.

Lakeguard is thereby fully transparent to users. Users simply run their workloads in Scala, Python, SQL and R on Databricks's compute, including interactive workloads that leverage Databricks' standard multi-user compute. Lakeguard allows customers to achieve significantly lower costs than filtering-based approaches [4] while supporting richer policies than file- and column-level approaches [3, 21]

The remainder of this paper is structured as follows. In Section 2 we explain in detail the challenges when implementing fine-grained data governance for enterprise data engineering workloads. In Section 3 we present the novel approach of Databricks Lakeguard by combining Spark Connect with sandboxing technology to achieve full multi-user capabilities. Furthermore, we outline how to efficiently offload fine-grained access control for workloads requiring privileged access to the compute. We then follow Section 4 in which we present how we integrate these capabilities directly into the Databricks data platform. Section 5 briefly analyzes the overhead of user code isolation. Section 6 describes how we take advantage of the fundamental architectural shift to further evolve the Apache Spark platform in completely new ways. We close this paper with a comparison of our approach with related work in Section 7 and conclude in Section 8.

2 Challenges enforcing fine-grained access control for enterprise workloads

When companies try to enforce fine-grained access control policies across their data and AI workloads, they face a number of challenges, e.g., managing access policies uniformly across different workload types or managing data privacy and security compliance requirements consistently. These challenges prevent them from adopting a unified set of policies and force them into fragmented governance rules that require administrators to reason about each system individually.

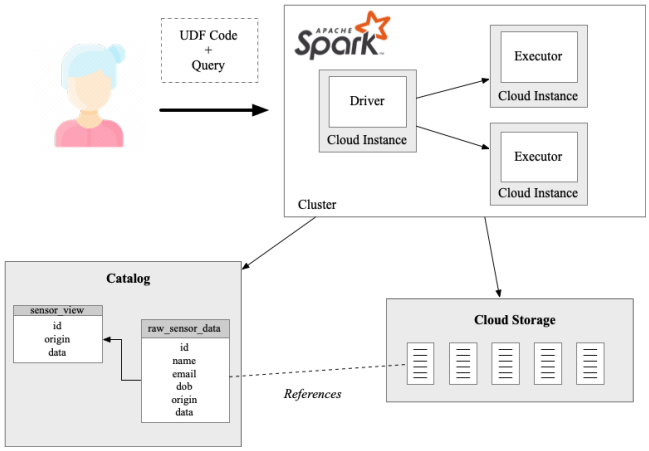

Figure 1: Illustrating the architecture and participating components in the motivating example.

2.1 Motivating example

An enterprise in the healthcare industry manages highly sensitive patient and clinical trial data. All data is stored in cloud storage, using Delta tables and Unity Catalog. Most of the table columns contain nonsensitive data, from simple relational data (e.g., location and timestamp values) to binary data (audio and video files), while others contain PII of patients, which must adhere to the strongest data protection requirements.

Processing data with Apache Spark allows users to perform fast analytical queries while including domain-specific logic such as user-defined functions (UDFs) in query processing. In our example, illustrated in Fig.1, the data science team extracts features from binary sensor data to produce a more suitable representation for further processing, using UDFs performing conversions. Data scientists have access to most of the sensor data but must not have access to PII. Therefore, the administrator created a dedicated `sensor_view` for the data scientists team, which filters out PII present in `raw_data_table`.

The administrator expects that any governance rule defined in the catalog are always enforced at runtime: when data scientists collaboratively work on the sensor data with SparkML for model training, when users perform ad hoc SQL analysis on the sensor data, or during the hourly data ingestion via Scala-based ETL pipelines.

Although fine-grained access control has been well established in data warehouses, there are a number of challenges when using these techniques for big data processing, which will be outlined in the following sections.

2.2 From cluster-bound to user-bound data access permissions

In traditional data warehouses, direct access to the raw data is not supported. Data access is always granted through catalog objects, such as tables and views. In data engineering and data science however, users typically get access to raw files to transform them into the desired representation. In fact, first-generation big data catalogs,

such as Hive Metastore, did not enforce data access, but only dealt with metadata management. Data access permissions were defined at the file or prefix level on cloud object storage. Clusters became the unit of isolation in terms of security and governance boundaries: data access was configured at the cluster level via storage access policies (e.g., AWS instance profiles) directly used by Spark. Users gained access to data *indirectly* via cluster access: as such, data access permissions were *cluster-bound*.

To enforce fine-grained access control, a common albeit inefficient practice was to create data replicas for specific user groups, where sensitive data were removed. Users would gain access to their filtered dataset through a dedicated cluster that had explicitly configured the necessary access credentials.

Creating and maintaining such replicas comes with a high operational burden: increased storage cost, staleness of copied data, and higher maintenance cost (e.g., GDPR across data copies), as well as provisioning dedicated compute/access credential per dataset. As a consequence, in second-generation data catalogs such as Unity Catalog and Iceberg REST Catalog, data access permissions are now an integral part of the catalog, moving data access from *cluster-bound* to *user-bound*. The catalog service ensures that each user can only access data to which they have been granted access. These permissions must be enforced at all times: for any data and AI workload and on any compute, as illustrated in Fig. 2. The Apache Spark driver accesses the table metadata by requesting it from the catalog, and the Spark executor nodes of the cluster request access to temporary credentials for the tables they are accessing. At all times, access to credentials is routed through the catalog and is always associated with the requesting user identity.

2.3 Enforcing fine-grained access control

The preferred approach for efficient data governance for any data and AI workload is to use well-established fine-grained access control mechanisms present in data warehousing, such as views, table-centric access policies such as row filters and column masks, and

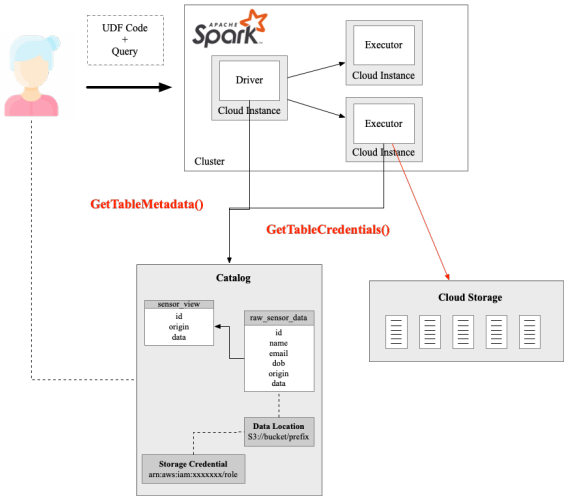

Figure 2: Moving storage access permissions from cluster boundaries to the catalog.

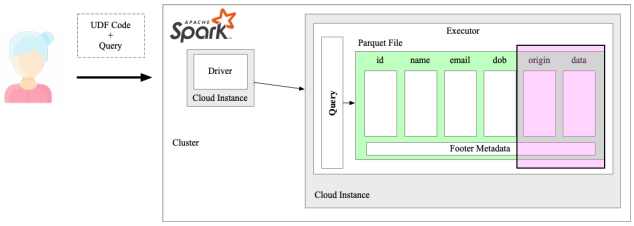

Figure 3: Dynamic fine-grained access control on cell-level

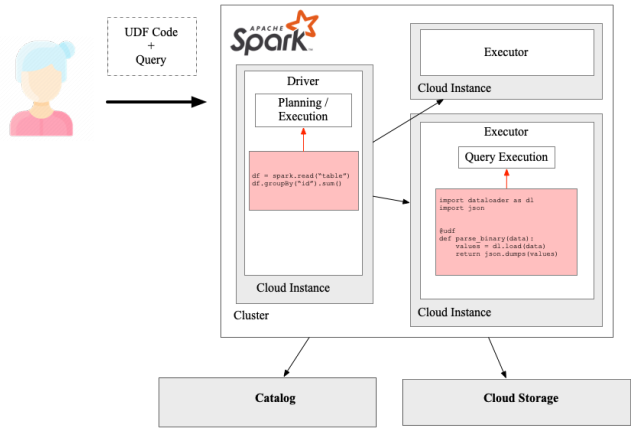

Figure 4: User code running together with the Apache Spark framework

dynamic access policies in the form of attribute-based access control (ABAC) [15].

A common example is to grant a user access to a subset of the rows of a table based on a SQL expression such as `CURRENT_USER()`. In this case, the engine reads and parses all the necessary data to be able to evaluate the expression. As cloud storage vendors manage access at the object level, for file formats such as Parquet, there is no easy way to restrict data access to read only the subset of bytes the user has access to (see Fig. 3). As a consequence, any user code (in the form of UDFs, or client application code) already present in the JVM is able to access this data using Python or Scala.

Although prior art [33] enforced fine-grained permissions by limiting the user surface to only SQL, this is not acceptable for a modern data and AI platform that also caters to data engineers and data scientists using Python, Scala/Java, or R.

To ensure that fine-grained access control is always enforced, in all languages when running Apache Spark workloads, we must ensure that user code is always isolated from the Spark engine so that no residual data is accessible to users running the application.

2.4 User code isolation

In contrast to SQL-based workloads in data warehouses, in systems such as Apache Spark, users have the ability to write their code in Scala, Python, R, and SQL. In an Apache Spark application, there are two main avenues of how user code is used. First, the application code is deployed directly with the Spark driver, providing the user with privileged access to the JVM responsible for metadata

processing, credentials management, and query processing. Second, Apache Spark allows users to extend and customize their data processing logic with user code, e.g., as UDFs running on Spark executors. These UDFs run in a shared JVM on executors with privileged access to the underlying machine. For example, a Scala UDF could write arbitrary files to the file system or ex-filtrate data to an external system. When UDFs are used in conjunction with fine-grained access control mechanisms, it becomes impossible to enforce these governance rules securely.

As a consequence, we have to make sure that any user code, be it UDFs running on Spark executors or user code running on the Spark driver, is fully isolated from the engine, with no access to the underlying machine and governed egress control.

2.5 Multi-user capabilities

With the scale of the number of different workloads where rich data governance rules that must be enforced, enterprises are looking for ways to simplify the organization and management of compute resources. Using more compute resources to enforce fine-grained access control is not desired because of increased cost and lower overall utilization, in particular, for interactive workloads. Customers and data platform administrators are no longer willing to provision compute for individual users to enforce governance but want multiple users to share clusters to increase utilization and reduce cost.

Since Apache Spark runs workloads in a single JVM, all users share the execution environment. As a consequence, malicious users can read residual state but also all the data Spark fetched to process other users' queries, leading to privilege escalation. As a consequence, to enable multi-user capabilities, we must ensure that user code is also isolated across users.

2.6 Summary

Table 1 summarizes the current status quo on how to integrate rich access policies uniformly on different platforms. Only Databricks Lakeguard uniformly enforces governance rules based on rich access policies across all workloads. Other platforms require integration with additional services to provide richer and more fine-grained access policies thereby increasing complexity and cost.

In summary, today, enterprises and organizations that want to adopt a unified approach for fine-grained access control across their data and AI workloads have to make sub-optimal choices.

(1) Avoid fine-grained data access control to continue being able to grant permissions at the cluster or file level.
(2) Provision compute for each individual user separately to be able to manage permissions for users independently. This means that permissions and access control are still tied to the compute instead of users. In addition, even in per-user clusters, fine-grained access control cannot be efficiently enforced.
(3) Implementing data governance policies separately for each use case, depending on the system used to access the data.

Any of these choices leads to the duplication of datasets to manage permissions at a finer level, an increase in the cost of more storage and compute, an increase in administrative complexity, and an overall higher overhead.

3 Databricks Lakeguard

To address the requirements of enterprise customers to provide a unified governance across all data and AI workloads, we introduce Databricks Lakeguard. Lakeguard is the first system to efficiently enforce coarse- *and* fine-grained data governance rules at runtime across single-user *and* multi-user workloads including declarative SQL-based workloads as well as data engineering and machine learning workloads in imperative programming languages. Lakeguard supports rich access control policies such as dynamic views, row-level filters, and column-masking rules familiar to data warehouse users.

In this section, we describe the four ingredients of Lakeguard in Databricks in more detail. In Section 3.1, we explain how Databricks' Unity Catalog acts as the central governance catalog. In Section 3.2, we describe the Spark Connect API and its components: the Spark Connect client, the Spark Connect protocol, and the Spark Connect service in more detail. In Section 3.3, we describe how user code is securely isolated from the core Spark engine. Finally, in Section 3.4, we show how governance is enforced even in case low-level machine access is needed using external fine-grained access control.

3.1 Governance using Unity Catalog

In Databricks, Unity Catalog is used to govern all of an enterprise's data. In contrast to the traditional Hive Metastore interface, Unity Catalog manages access to tabular data, but also cloud storage paths. Unity Catalog is meant to provide exactly one way to govern data access: the flexibility of choosing between path-based and table-based access is important to support the diverse set of workloads of our customers using Apache Spark and data warehouses, especially when ingesting and transforming raw files.

In addition to tabular data and files, Unity Catalog is the central component of the Data Intelligence Platform [11]. It governs access to machine learning models, feature functions, UDFs in SQL and Python, and many other securable objects.

We recently chose to open-source Unity Catalog and are advancing the features of the open Unity Catalog in collaboration with the Open Source community. This process further enhances its capabilities according to the needs of our users, and the community at large. The open source Unity Catalog, together with the open file formats Delta [9] and Iceberg [6], is crucial in supporting Databricks' Open Lakehouse strategy.

3.2 Spark Connect API

When running data and AI workloads in Apache Spark, the application code (e.g., the notebook) shares the same JVM with the rest of the engine, as illustrated in Figure 4. To decouple the application code from the trusted engine code, we contributed Spark Connect to the Apache Spark project.

Apache Spark offers multiple APIs for data access: RDD [34], DataFrame, and Dataset [10], while the recommendation is for users to use the declarative DataFrame and Dataset APIs. Today, new applications are predominantly using the DataFrame API, as it provides the greatest potential for the core data processing engine to optimize operations [10, 31]. An application using the low-level RDD APIs is a black-box to the query optimizer, and thus can even

Property	Databricks Lakeguard	AWS EMR Membrane[19]	AWS Lake Formation	Microsoft Fabric One Lake (Spark)	Google Dataproc with BigLake[16]
Governance					
Unified Policies for DW and DS/DE	✓	✗	✗	DWH Only	✓
Catalog UDFs	Python	✗	✗	✗	BigQuery Spark Stored Procedures
User Code *executed as part of the workload, application code and UDF*					
Single User	SQL, Python, Scala, R	SQL, Python, Scala, R	n/a	SQL, Python, Scala, R	SQL, Python, Scala, R
Multi-User	SQL, Python, Scala, R	✗	n/a	SQL (DWH Only)	✗
FGAC methods supported					
Row-Filter	✓	✓	✓	✗	✓
Column-Masks	✓	✓	✓	✗	✓
Views	✓	✓	✗	✓	✗
Materialized Views	✓	✗	✗	✗	✗
External Filtering	✓	✗	✓	✗	BQ Storage API

Table 1: Comparing data platform governance solutions that leverage and can be used with Apache Spark.

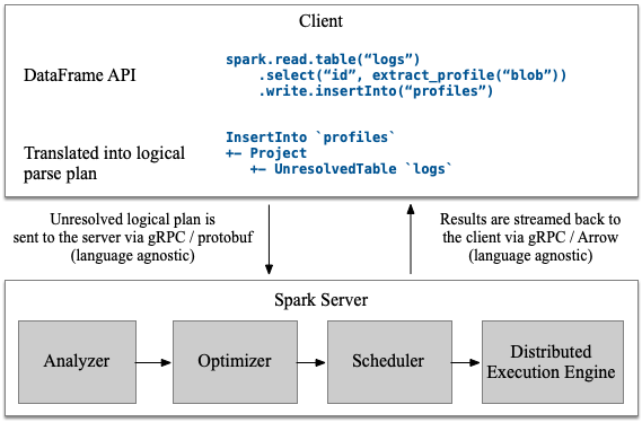

Figure 5: Spark Connect Flow

result in worse performance. The general recommendation is to only use these low-level APIs for very specific and highly optimized use cases.

Based on the observation that the programming interface has become Spark's logical plan for new Spark applications, we proposed decoupling the client application from the Spark engine based on this interface. However, instead of directly depending on the specific Scala interface, we introduce a generic abstraction that is language-independent and version-compatible using an externally defined protocol based on the Google Protocol Buffers format [13]. The separation of concerns between the client application and the Spark server has additional benefits: Spark Connect provides a backward-compatible interface that makes it easier to upgrade Spark versions without breaking compatibility for client applications and provides additional stability to the Spark cluster by decoupling the failures of the client application from the server.

The general execution flow of a Spark Connect query is described in Figure 5: First, when the client application calls the Spark DataFrame APIs, we capture these operations. When an operation on the plan is executed, we translate the chain of operations into the Spark Connect Protocol Buffers format. The query plan is sent to the Spark Connect service, which runs alongside the Spark Context. The Spark Connect service implements a Google gRPC [14] service interface. Here, the plan is deserialized and translated into Spark's logical plan structure. We then call the command on the plan, for example, `collect()`, which triggers the regular processing logic of Apache Spark. Once the query execution, with its regular stages for analysis, optimization, and query execution is done, the Spark Connect service serializes the result rows into Arrow IPC messages and streams them back to the client. The client can now transform the Arrow IPC message stream into the native representation of its choice.

The Spark Connect architecture consists of three core components:

(1) The language-agnostic client implementation
(2) The Spark Connect protocol buffer format
(3) The Spark Connect service on the Spark driver

The client communicates with the Spark Connect service using the gRPC message format over an HTTP/2 connection. In the following sections, we will explain how the different components interact with each other in more detail. Leveraging a standard HTTP connection has additional benefits, as it allows platform providers to seamlessly integrate with their own authorization and proxy infrastructure.

3.2.1 Spark Connect client. The role of the language-agnostic Spark Connect client is to provide the convenient, Spark-idiomatic DataFrame implementation to interact with the Spark server. Currently, there are already a few implementations of the Spark Connect protocol available, namely, in Scala, Python, Rust [25], Go [23], C# [22] / .Net [24]. For developers familiar with the Spark DataFrame API,

using these clients provides low cognitive overhead as they follow the same design patterns but are adapted to the programming language-specific idioms.

In practice, this means that the Spark Connect client translates the API operations on the exposed DataFrame API into the appropriate protocol buffer representation.

When executing an action, the Spark Connect client will send the protobuf representation from the client to the server and wait for the responses to be received. The response data (serialized Arrow row batches) are then passed on directly to the caller for zero-copy result extraction or translated into programming-language-specific native values.

3.2.2 Spark Connect protocol and RPC. The service definition of the Spark Connect protocol is centered on the primary query execution API called `ExecutePlan` / `AnalyzePlan` and additional control RPCs to manage the execution. The root of any execution is either a logical relation or a command. In Spark Connect, we distinguish between relations and commands to indicate composability and side effects. Relations do not have side effects when executed and can be composed, whereas commands are not composable and can have side effects when executing them. Relations are modeled after high-level logical relational operations like `join` or `sort` following the example of Spark's DataFrame API. Lastly, the protocol captures Expressions that transform individual columns of Relations.

The Spark Connect execution mechanism combines synchronous and asynchronous query execution to reduce latency for short-running operations and simplify the implementation of clients in other languages. In addition, the execution protocol supports the different idle connection termination strategies of cloud load balancers and interrupted connections from remote clients. These properties have proven to be very important based on our experience of running Spark Connect workloads in production as part of Databricks Connect [12].

In traditional Spark applications, extensibility is achieved by directly inheriting from public developer APIs in Spark. With the separation of the client application from the main Spark JVM, Spark Connect needs to provide an additional way to support extensibility. Therefore, in Spark Connect, all major interfaces for relations, expressions, and commands provide explicit extension points. The extension points provide a mechanism to transparently embed custom message types as part of the execution. This allows Spark users to develop and install Spark Connect plugins that provide embedded functionality directly as part of the core Spark Connect protocol without having to modify the protocol explicitly. Similarly to the overall benefits of Spark Connect, separating the plugin implementation of the client provides a stable interface between the client and the server and makes it easier for plugin providers to support clients through new Spark versions. As long as plugin providers follow the same compatibility requirements as defined for Spark Connect, their users can benefit from the same stability and compatibility benefits. A prominent example is the Delta [9] extension for Spark Connect, which provides the custom relation and command types needed to support Delta-specific API commands for Spark Connect clients.

3.2.3 Spark Connect service. The Spark Connect service provides the implementation of the gRPC service based on the protocol buffer definition. In its initial version published in Apache Spark 3.4 the service was implemented as a Spark plug-in that was loaded during the startup of the Spark context. With the current development of Apache Spark 4.0, the Spark Connect service is now a core component and will be directly embedded into the core Spark distribution.

The main responsibility of the Spark Connect service is to manage incoming connections and map them to individual Spark Sessions. A Spark Session encapsulates all client-specific application state like, for example, registered temporary views or functions. In addition, the service is responsible for managing the life cycle of these Spark sessions and the temporary state that is attached to them. Lastly, the Spark Connect service manages the life cycle of concurrent query executions by regularly checking if the clients continue to reattach to the query, the overall query makes progress, and if a client disappears, abandon and tombstone the query execution.

In Databricks, we have significantly enhanced Spark's session management, authentication, and authorization capabilities to provide a full multi-user system. This allows us to not only securely share the underlying compute infrastructure but also full auditing of all individual user actions. With this secure integration, Spark Connect is tightly integrate with Databricks' Unity Catalog governance platform.

3.3 Isolating user code from the engine in Apache Spark

With the client code being separated from the engine, it is also necessary to isolate the user code from the engine that is used directly during query processing. In Section 2, we outline in the example how a specific user code is used during data and query processing. In contrast to traditional SQL-based database systems, user code is not provided simply as a declarative SQL expression, but the users of Apache Spark are writing their domain-specific user code in Python, Scala, or even R. Importantly, the user code executed in Apache Spark is not just ephemeral code submitted by the user during query execution, but also code that is governed by Unity Catalog. Python UDFs are cataloged objects, similar to Hive UDFs, allowing customers to reuse domain-specific user code across all of their data and AI workloads in a secure way.

A typical example is imperative logic that needs to interact with external systems and convert data appropriately, as shown in Figure 6. Here, a Python UDF is used to call an external service that returns air quality measures based on a zip code. Previously, all user code was executed directly in the engine's JVM, allowing users to access all data, credentials, and secrets processed in the environment.

To overcome this limitation, we had to make major modifications to the query execution engine of Apache Spark. First, we built the infrastructure to be able to securely execute user code in containerized sandboxes outside of the Spark engine. Second, we modified existing query operators and added new query operators that handle containerized execution of user code.

In Figure 7, we outline the abstract architecture of a host of a cluster in the Databricks environment. The host is provisioned into a runtime environment that is accessible by Apache Spark, and a

```python
@udf(returnType="float")
def resolve_zip_to_air_quality(zip):
    resp = requests.post(
        f"http://example.aqi.com/zip/{zip}")
    return float(resp.json()["yesterday"])

data = spark.table("customers")
data.select(
    resolve_zip_to_air_quality(col("zip"))).show()
```

Figure 6: Example of PySpark UDF calling an external service.

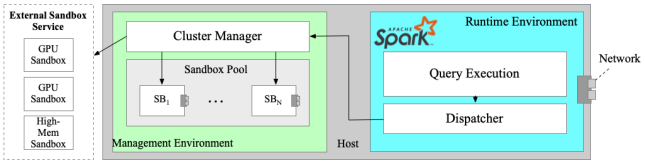

Figure 7: Databricks Host Architecture with Cluster Manager

secure and protected cluster management environment that is fully decoupled from the Apache Spark processes.

When user code is executed as part of query processing, the query process interacts with the component called `Dispatcher` that manages the sandboxes for query processes and users running concurrently in the cluster. When a new sandbox is required, the dispatcher requests a new sandbox from Databricks' cluster manager. The cluster manager creates a new sandbox and proxies all traffic between the Apache Spark process and the sandbox itself. The sandbox runs fully isolated from the runtime environment and is not connected to it directly. We use dynamically controlled network rules and Linux kernel network namespaces to additionally control the egress network traffic of the UDF.

To reduce overhead and optimize query execution, we have introduced several query optimization rules that collapse and fuse the execution of user code into as few sandboxes as possible. In addition to ephemeral user code written as part of the current session, Unity Catalog catalogs user code in the form of functions, for example, Python UDFs. These functions can be used across workloads. This requires the dispatcher to implement and manage different trust domains of user code executed in the context of a single user query. A trust domain groups all user code that is owned by the same user. However, cataloged UDFs may be written by different users. Therefore, isolation is necessary to prevent side-channel attacks and data leaks on the runtime environment of code running inside a sandbox.

The trust domains become pipeline breakers for the optimization rules we use to fuse code together for efficiency and ensure that the execution is free of side effects.

The simple and efficient design of the infrastructure allows us to run user code not just on the Apache Spark executors but on all nodes of the cluster. Furthermore, the cluster manager interface to create sandboxes allows us to generalize the concept of executing user code. For requests that have specific resource requirements, for example, GPUs or significant amounts of memory, we can route these requests to specialized execution environments outside of the cluster.

3.4 External fine-grained access control

For some workloads, having full privileged access to the underlying machine is a requirement. A typical example are machine learning workloads that use one or more GPUs as part of their operations. Using GPUs requires direct memory access, direct access to the device drivers, and privileged access to the network for the distribution of work. In this environment, it is not possible to run user code completely separated from the engine. Previously, this meant that customers who wanted to enforce their rich governance policies uniformly across all kinds of workloads would have to create use case-specific copies of data. In turn, administrators would have to track these replicas and account for them in their data governance rules.

Databricks Lakeguard solved this challenge by providing automatic external fine-grained access control (eFGAC) for workloads running on compute types with dedicated access to the cluster and Apache Spark system.

To do so, we track the security and execution properties of each cluster when communicating with Unity Catalog through privilege scopes. This is necessary because the individual user might have access to the entities governed by Unity Catalog, but the execution environment is not able to properly enforce the governance policies.

To illustrate the behavior, let us assume that the user on a dedicated cluster wants to execute a simple query on a table `sales` with a row filter that dynamically restricts access to only sales in the `US` for this particular user.

```sql
SELECT amount, date, seller
FROM sales
WHERE date = '2024-12-01'
```

When this query is parsed, it is translated into the logical plan and the fully resolved logical plan, as shown in Figure 8. In our implementation of row filters, the query planner automatically injects a `SecureView` into the query plan that prevents the propagation of unsafe expressions to avoid leaking data.

However, on clusters with privileged access, the cluster can only fetch the basic metadata about the involved relations but not details about the predicates or literals used in the row-filter expressions. In addition, the metadata for the participating relations is annotated to indicate that these objects cannot be processed locally.

Instead of translating the query from the source query to the fully resolved query shown in Figure 8, it is dynamically rewritten to perform the fine-grained access control externally. This is highlighted as the rewritten query in Figure 8.

The foundation for eFGAC is Spark Connect (Section 3.2 and the Databricks Serverless Spark offering (see Section 6.2). The core principle of the query rewriting strategy for eFGAC is that it operates at the highest logical level in the query plan. During the rewrite phase, we identify the relations that need to be processed externally and insert a remote scan operation in their place as a leaf node in the plan. However, these leaf nodes are not simply scan operators, but we leverage the optimizer rules to push additional refinements into the remote scan. Most importantly, this allows us to push partial aggregations, projections, and filters into the remote scan. It is important to mention that the interface of the eFGAC scan does not need to be aware of any specific governance rules. The eFGAC scan operates on the unresolved logical plan level only.

Figure 8: Translating and rewriting a source query with fine-grained access control into a remote filtered data source operation.

After the query rewrite phase is finished, during query execution, the sub-query is submitted to Serverless Spark for execution. It is important to mention that at no point in time, resources have to be manually allocated or provisioned: all of the assignment and work executing happens completely transparently to the user and is fully managed by Databricks.

On the serverless endpoint that receives the eFGAC subquery, the incoming Spark Connect plan is parsed, analyzed, and optimized as any other Spark query. This means that during query planning and optimization, the cluster fetches the metadata from Unity Catalog and inserts the row filter of our example again into the query. For partial aggregations, filters, and projections submitted as part of the query, they are simply handled as part of the regular query execution.

The Serverless Spark endpoint supports two result aggregation modes that are chosen, for example, based on the size of the result set. If the result is small, the results are returned inline with the query to the origin cluster to avoid additional latency. For larger result sets, the intermediate data is persisted in parallel in cloud storage and processed in parallel on the origin cluster.

In contrast to other solutions such as AWS LakeFormation or Redshift Spectrum, our approach overcomes an important limitation: the ability to handle more than basic scans. eFGAC in Databricks is capable of processing all queries supported by the Spark Connect protocol. This is important as it shows the universality of the approach. In Databricks, eFGAC allows reading from simple tables that have comprehensive data governance policies, such as row filters and column masks, as well as from complex views, including materialized views.

Lastly, it is important to mention that eFGAC is not only usable in the context of Databricks clusters but can be used seamlessly from any external engine like Presto/Trino or other Spark distributions to enforce data governance. This particular, limited, use case is in spirit similar to the Google's BigLake approach [16].

4 Unity Catalog compute in Databricks

In the previous sections, we have explained in detail the building blocks that are necessary to provide unified, cost-effective, and secure data processing capabilities in Databricks. As described, the application programming surface when using Spark Connect with the isolation primitives is slightly different from the traditional Apache Spark surface. For our customers, it is important that Databricks provides a stable and consistent environment that gives customers the necessary time to choose when to upgrade to the new architecture but, at the same time, is able to enforce the data governance rules.

Consequently, governance must not only be enforced on the compute itself, but Databricks' Unity Catalog must reason about the source of the requests coming from clusters. Due to the way that Unity Catalog is integrated into the authorization and authentication system in Databricks, every caller authenticating to Unity Catalog has additional information about the credential scope of the caller. When the Apache Spark engine requests access to cloud storage credentials from a specific compute type, Unity Catalog is able to identify the individual identity and compute type that made the request. This behavior is a crucial aspect in enforcing security and governance across the platform because it acts as an additional layer of depth to make sure that individual users only have access to the data that they are supposed to.

Therefore, Databricks offers two compute types to its customers: Standard and Dedicated compute.

4.1 Standard clusters

The primary compute type in Databricks is the standard cluster. This type of cluster unifies all the building blocks to provide a fully secure multi-user-capable environment that utilizes the power of Apache Spark. Standard clusters can be used for scheduled and ad hoc job submission and are fully supported throughout the platform. In contrast to other vendors, the multi-user capabilities of the standard cluster allow us to securely share the cluster between many users and enforce all individual users' permissions. All of the data governance capabilities (dynamic views, row filters, column masking) are directly enforced on standard clusters on top of the coarse-grained object-level access control.

Our customers can use SQL, Python and Scala to develop and schedule their workloads, making it the only system that is able to have multi-user access to Spark across all of Spark's main programming languages.

In standard clusters, all Spark client applications run fully isolated and sandboxed on the driver, and user code runs isolated and sandboxed on the executors. Users who attach and run code on the cluster do not have direct access to the core engine. Temporary credentials, encryption keys, and other application-specific states are not accessible directly to any user. This complete decoupling of the application user from the engine allows the Databricks standard cluster to process data in a privileged fashion and to execute row filters and column masks locally.

Many of the jobs and applications written for Spark use additional libraries for processing. As part of the isolation primitives on standard clusters, we additionally isolate how, for example, Python dependencies are managed and installed. This allows us to track individual dependencies for each individual user when connected to a single cluster. Each of the environments specified dynamically by a user is separate and never shared with the core processing engine.

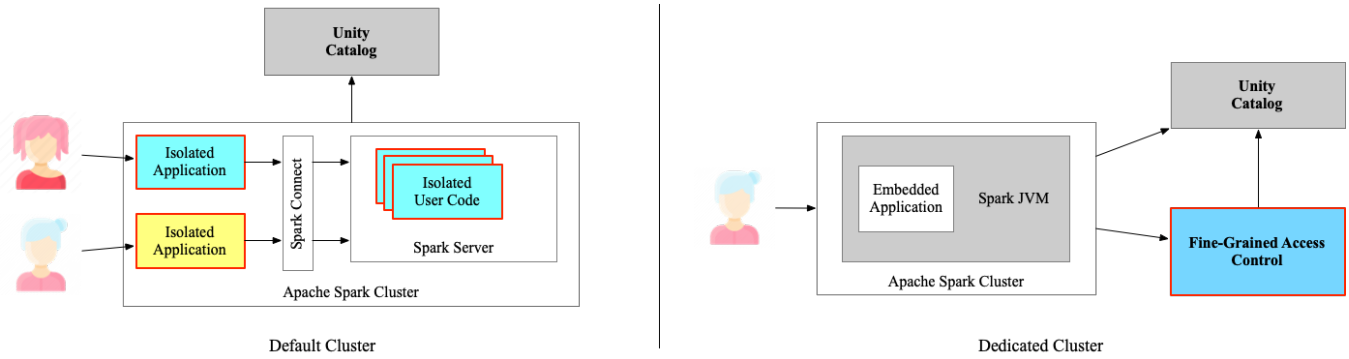

Figure 9: Fully governed compute types in Databricks leveraging the Lakeguard Architecture,

However, we support administrators in making conscious choices about installing additional libraries on the cluster that interact directly with the core Apache Spark engine. To ensure the intent and security of the clusters, we have established a configuration process that requires the delegation of explicit intent from both workspace and cluster administrators about which explicit libraries should be made available to the core engine.

4.2 Dedicated clusters

For customers who need access to the underlying virtual machine, for example, to work with one or more GPUs, direct network access, and access to special device drivers in addition to the Apache Spark engine, Databricks offers dedicated clusters. Due to the low-level cluster access, Dedicated clusters can only enforce coarse-grained access permissions, so we provide our customers with automatic access to external fine-grained access control (cf. Subsection 3.4). The benefit is that this does not require any configuration, but is integrated seamlessly to provide access to tables with row- and column-level access control and views.

The most important distinction between dedicated and standard clusters is that dedicated cluster cannot be shared by more than one identity. As multiple users bring individual permissions to such a cluster and every user has the same privileged access to the host, it is not possible to enforce them independently.

The benefits for our customers are that dedicated clusters provides a convenient way for them to onboard to the features offered by Unity Catalog and the Databricks platform in general. From our analysis of customer usage, the main use cases for dedicated clusters are workloads for machine learning, both for model training and inference, and second, mature workflows that have been relying on traditional low-level Spark APIs and therefore rely on the processing of data with RDDs [34].

For security reasons, it is not possible for multiple identities to share a single dedicated cluster. This increases operational burden and cost for our customers in particular for interactive use cases when multiple users want to share and collaborate on designing data processing logic or machine learning experiments. To alleviate the immediate problems of sharing interactive clusters, Databricks offers the ability for multiple identities to share a single dedicated cluster for members of the same access group. When attaching to the cluster, all individual users' permissions are reduced to exactly the permissions of the group. This dynamic down-scoping of permissions allows us to retain the originally connecting user identity and guarantee that all attached users have exactly the same permissions.

Num UDF	Simple UDF Sum(a+b)	Hash UDF 100× SHA256
1	9.53%	3.37%
2	8.44%	4.29%
5	11.19%	4.77%
10	12.02%	4.15%

Table 2: Comparing the relative worst-case overhead of executing Python code in a sandbox with unisolated code in Databricks.

5 Evaluation

Previously, running user code using different isolation primitives has been evaluated in different contexts, for example, in [5] or [20]. Since the properties of isolation techniques like gVisor have been explored before, we focus on a very basic validation of these experiments in Databricks' production environment.

For the experimental evaluation of our approach to isolating user code from the engine, we have to consider two major dimensions. The first dimension is the static overhead for running isolated code in our sandbox environment. The second aspect is the continuous overhead that a UDF incurs because the code is running in isolation. In addition, the experiment is designed to validate that fusing UDF calls into a single sandbox can help prevent additional runtime overhead.

We tested continuous overhead using two simple experiments; we compare the execution time of a Apache Spark query over a fixed number of rows and evaluate a UDF per row that simply returns the sum of two arguments. Since computing the sum of two values in itself does not consume meaningful resources, it highlights the potential worst-case scenario in which the overhead is dominated by moving batches of records into the sandbox and returning it.

In the second experiment, we compare the execution time of a UDF that calculates a SHA256 checksum with 100 iterations. The goal of this experiment is to identify the pure CPU overhead of running performance-sensitive code inside the sandbox. The execution environment was a 2-node Databricks cluster using the Databricks Runtime version 15.4 LTS on AWS using in total 3 r6id.xlarge instance types. We executed the same code multiple times on both Standard and Dedicated Databricks cluster types.

In Table 2 we present the results of these experiments. For the first case in which the UDF simply returns the sum of its arguments, the measured overhead compared to running the code without isolation is ≈ 10%. For the second experiment, the time spent in the UDF code per row is significantly higher. In this case, the overhead is reduced to ≈ 4.8%. The results confirm that running user code in isolation comes at the price of slightly increased latency. However, from our experience in analyzing customer workloads, we see that the contribution of, for example, Python UDF runtime on the overall query runtime is typically only a small fraction because Python code itself is significantly slower than the highly optimized query execution code in Databricks Spark. Lastly, we can observe that our approach to fusing multiple UDF executions for a single row works, and increasing the number of UDFs does not have an outsized impact on the overall latency.

The last aspect of our performance evaluation was the impact of the sandbox startup time. The sandbox startup time is split into two categories: First, provisioning of the sandbox and starting the Python process inside the sandbox. In our experiment, we see a maximum duration of cold start in all experiments of ≈ 2*s*. However, this latency occurs only for the very first Python UDF across the whole user session. Subsequent query executions reuse the already existing sandbox, and the overall startup cost is quickly amortized.

6 Architectural evolution

The introduction of Spark Connect to Apache Spark allowed the evolution from a cluster-centric data processing framework to a fully decoupled client-server architecture. This decoupling allowed the implementation of features and improvements that allow Spark to be used in completely new ways and improve the overall user experience in many ways. In this section, we will highlight how Spark Connect fundamentally changed the way Databricks unifies data and AI workloads for Apache Spark.

6.1 Databricks Connect

The starting point of the evolution is the first customer-facing product that uses Spark Connect commercially. For many of our customers, interactively developing and debugging their data engineering workloads in an IDE is a critically important task. Previously, these users were forced to develop their workloads that use Apache Spark locally by running a local version of Apache Spark and then later deploying the code in Databricks and verifying it as a scheduled job. This mismatch between developing code locally and verifying it later asynchronously in Databricks makes it harder to establish fast inner development loops and software engineering best practices.

But since data, governance, and applications are managed in Databricks, Apache Spark running locally could never fully replace the ability to verify the workloads directly running on Databricks.

With Spark Connect, there is an established remote protocol to communicate between the code running locally and the Spark operations running remotely. Databricks Connect is a thin layer on top of the native Spark Connect protocol implementation in PySpark that manages authentication and authorization with the Databricks compute endpoint. The rest of the specific protocol semantics are the same as in open source Spark.

Databricks Connect allows customers to connect from anywhere to Databricks and directly run their Apache Spark workloads from their local machines. It allows connecting to designated clusters, both Standard and Dedicated, and Databricks Serverless.

Databricks Connect not only enhances developer efficiency, but also enables the integration of external applications with data managed in Databricks in a completely new manner. Lastly, since Spark Connect provides a language-agnostic way of connecting to Spark, it allows Spark Connect protocol implementations in other languages to connect to Apache Spark and Databricks remotely. Currently, there are implementations for Spark Connect in many different languages such as Rust [25], Golang [23], C# [22] and .NET [24]. Each of these open and independent client implementations can directly connect to Databricks in the same way as Databricks Connect does.

6.2 Databricks Serverless Spark

When using Apache Spark, users rarely question the need to provision a cluster or at least an endpoint. In the context of many vendors, the term "serverless" simply means that the compute hardware is abstractly managed by the provider, but overall the cluster is still a present concept because the application and execution have always been very tightly bundled together. The lack of user isolation in these offerings does not allow efficient resource sharing, increasing cost and operational burden.

In Databricks, we changed the way Apache Spark workloads are run for our serverless spark offering. First, we introduced a fully decoupled architecture in which the client application is physically separated from Spark. This allows us to only provide compute resources to the client application when Spark is really needed. Second, all of the managed compute running the Apache Spark workloads are based on the previously introduced Standard cluster architecture, providing full multi-user capabilities and allowing us to share resources between users in the same organization securely.

This allows us to manage the resources for the Apache Spark workloads in a much more efficient way. As Databricks completely manages the resource provisioning, we observe and analyze similar workloads to understand and predict their resource requirements. The knowledge about past and future workloads feeds machine learning models that we use to automatically scale an individual cluster or dynamically provision a new cluster for new incoming connections.

Furthermore, we enhanced the life-cycle management of Spark Sessions in Databricks' serverless Spark to be able to seamlessly migrate individual user sessions from one backend to another without incurring user visible downtime.

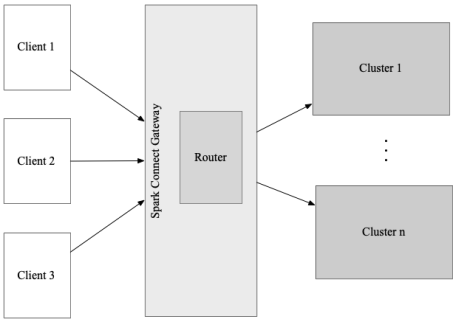

Figure 10: Basic Serverless Spark Architecture with a Workspace-Wide Endpoint

Figure 10 illustrates how clients connect to Databricks' serverless Spark offering. The client applications can be interactive experiences like notebooks or scheduled workloads like jobs. All workloads connect to the same endpoint per workspace. The workspace-level request is sent to the regional Spark Connect Gateway service that tracks resource management and current utilization for individual workspaces. Based on load and historical knowledge, the Spark Connect Gateway will now either provision a new cluster or forward the request to an existing cluster.

6.3 Versionless Spark workloads

One of the main goals of designing the Spark Connect protocol was to define it in a way that allows us to completely decouple the client-side implementation from the server. As we leverage the inherent properties of the protocol representation using the Protocol Buffer format, it is very easy for older clients to connect to newer server versions. As a result, we are able to achieve full backward compatibility for clients. In internal surveys with Databricks customers, we have identified that upgrading to more recent versions of the Databricks runtime is a big pain point. The challenges of updating versions arise from two areas: First, client applications that compile against Databricks' Spark version might have to deal with changes in the APIs that the client is using. While Apache Spark attempts to maintain binary compatibility in minor versions, in major versions changes can be larger. Second, the Databricks runtime environment, together with Apache Spark, bundles a set of libraries that the customer has to be aware of. If these dependencies conflict with the customer's code, it can lead to breaking previously running workloads. Customers prefer not to invest in workloads that are running successfully and prefer stability over new features.

In Databricks, we have addressed these challenges with Serverless Spark. We have isolated the client environment into an abstract *Workload Environment* that is versioned and includes, next to the specific Databricks Connect version, the set of dependencies that are bundled with the Databricks runtime. The *Workload Environment* is responsible for providing a stable surface for client applications and jobs that our customers can rely on.

When the client notebook or job connects to Serverless Spark endpoint, the Spark Connect protocol fully supports queries and executions from previous versions. This version independence is already enforced in open source Spark to make sure that the clients remain backward compatible.

There is a special case where client applications submit user code as part of the workload and query execution that depends on the libraries and dependencies present in the *Workload Environment*. In Databricks, we are able to leverage the same infrastructure that we have built to support user code isolation to support clients connecting to Spark using different Workload Environment versions. During the execution of the user code, the system will explicitly load the given workload environment and execute the user code exactly in this environment.

Explicit versioning of the workload environment has proven extremely successful in upgrading dependencies. For example, it allows us to support customer-specified code to be executed, isolated, next to the core engine using multiple different versions of the Python interpreter without the customer having to reason about the version of Serverless Spark. The most important requirements for customers are to be able to have a stable set of Spark APIs and a fixed set of dependencies and their versions.

7 Related Work

For a long time, enforcing fine-grained permissions on big data was not easily possible or only available with a very limited user surface (e.g., SQL only). In addition, systems like Apache Spark are typically one part of the overall data platforms and, as such, middleware systems like Apache Ranger [8] are trying to provide an entry point to managing permissions and access. However, such systems do not provide a way to enforce permissions directly in Apache Spark.

There are other approaches that integrate access and permission management directly into the Apache Spark engine. *GuardSpark++* [32, 33] is an example. The system implements purpose-based access control directly into the Spark planning and rewriting layer. However, this approach does not outline how to protect the assumed trusted engine from the user code. This means that the system still has to fall back to untrusted behavior when not using SQL.

The system closest to the capabilities that our system provides is the implementation of Amazon Web Services for their Apache Spark offering called *Amazon EMR Membrane*. Following our work on Spark Connect in Apache Spark in the open source community, the proprietary Amazon Membrane project [19] also recognizes that executing user code directly in the core processing engine creates a data governance and security issue when trying to enforce fine-grained access control. The approach of Membrane is to split an Apache Spark cluster into two security domains. A core trusted engine domain and a user code domain. Data exchange between domains occurs via shuffle operations. The biggest drawbacks of this technical approach can be explained as follows: First, dividing the cluster into two security domains does not efficiently allow the sharing and scaling of resources based on need. Both domains have to be effectively scaled and managed independently as they can never overlap due to potentially residual data and state remaining on the instances. This means that for highly variable workloads, the overall utilization of the cluster will be much lower, leading to increased costs for the customers and slower overall performance. Second, relying on a custom, specific serialization format of the

internal Apache Spark execution plans, the application code is not sufficiently independent of the rest of the Apache Spark cluster, making it difficult to upgrade to new versions of Apache Spark. Third and most importantly, while the architecture allows one to enforce fine-grained access control locally, it does not make Apache Spark fully multi-user capable. The current architecture does not foresee the ability for multiple users to share the cluster, thus further increasing the utilization of the resources and lowering costs. Similarly, Google's BigLake [16] proposes a way to integrate data managed in open file formats both in BigQuery directly and in Apache Spark through the BigQuery Read API. However, in the context of Apache Spark workloads, it is not directly integrated but is built on a similar approach to what is described in Section 3.4 with similar limitations and no support for processing of views.

When it is not possible to directly enforce fine-grained access control and permissions in a Apache Spark cluster, users are usually forced to employ external systems for data preprocessing outside of the cluster. This allows us to avoid implementing governance rules directly in the core engine. *AWS LakeFormation's Data Filtering* offering [4] provides these capabilities and allows external systems, not limited to Apache Spark, to integrate. However, as of today, it only supports simple scans and expressions. The *additional* external filtering capabilities that Databricks employs for Dedicated Clusters support full subquery execution with aggregations, projections, filters, and limit.

When we started to work on Spark Connect with the open source Apache Spark community, we looked extensively at different options of how to represent the protocol interaction between the client and the server. For example, we looked at using the Substrait [18] for the query plans sent from the client. There are two reasons why we did not choose Substrait for the query plan representation. First, the Substrait project started around the same time and it was not immediately clear if the direction of the project was to focus on logical or physical database query plans. Ultimately, Substrait is designed to handle both abstractions equally. In Spark Connect, however, we operate only on unresolved logical plans and unresolved expressions that are much simpler to represent and express. Since the Spark Connect representation is effectively a subset of the capabilities of Substrait, it is possible to translate a Spark Connect query plan into a Substrait-based representation. Although Spark Connect can support additional use cases similar to Apache Livy[7], the approaches are very different. Apache Livy essentially operates on plain text commands without any additional knowledge of the semantics of the executed operations.

The topic of isolating untrusted code has been extensively investigated in the past years. The most important research contributions stem from the work of the hyperscalers to efficiently and securely support lightweight function handlers. The most prominent implementation is Amazon's Firecracker [1] technology. Firecracker supports a very fast and almost native execution model; however, it requires direct native access to the instances running the Firecracker infrastructure. In Databricks, we deploy clusters to standard cloud VMs, and using bare-metal instances is not feasible. The overhead of running user code in higher-level virtualization solutions have been analyzed by Anjali et al. in [5]. Alternatives are to write custom application-specific isolation logic directly using Linux Kernel Namespaces and Seccomp [17]. However, manually constructing these rules is only possible for a very narrow application interface.

In [20], the authors have looked at a similar technology to isolate UDFs in a sandbox environment for Apache Spark. The primary difference from the work presented in this paper, the sandboxed environment is local to the cluster processing the data and avoids additional network interaction. In addition, it is not limited to registering UDFs in the catalog but supports the full surface of inlined UDFs in Spark.

8 Conclusions and future work

In this paper, we have shown how to uniformly, efficiently, and securely support fine-grained access control and multi-user capabilities on the Databricks platform for enterprise workloads leveraging Apache Spark. The approach based on using Spark Connect for client isolation and the proprietary user code isolation layer has proven to be very successful, supporting thousands of customers across the Databricks platform. In addition, this architecture started an architectural shift for Databricks Apache Spark workloads, where previously users and integrators always had to think about provisioning compute when trying to leverage Spark, our Serverless Spark offering has changed the way internal and external users build upon our platform. Most importantly, governance and security are always enforced and managed centrally by Unity Catalog.

We believe that in the future, there will be a strong adoption of Spark Connect in the context of Apache Spark workloads, not just in Databricks, but across the open source community as well. In the same way that JDBC interfaces have standardized access to databases, we believe that Spark Connect provides a similar abstract integration layer.

9 Acknowledgments

We appreciate the support of the open source community. The work would not have been possible without the dedication and effort of the Apache Spark community in supporting major evolutions of the open source project like Spark Connect. We would like to thank, in no particular order: Weichen Xu, Akhil Guedasa, Niranjan Jayakar, Nemanja Boric, Garland Zhang, Robert Dillitz, Xi Lyu, Biruk Tesfaye, Changgyoo Park, Alex Khakhlyuk, Haiyang Sun, Dmitry Sorokin, Vsevolod Stepanov, Ben Hurdelhey, Xinrong Meng, Haejoon Lee, Rui Wang, Ruifeng Zheng, Julek Sompolski, Tom van Bussel, Pengfei Xu, Bobby Wang, Allison Wang.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[2] Rakesh Agrawal, Paul Bird, Tyrone Grandison, Jerry Kiernan, Scott Logan, and Walid Rjaibi. 2005. Extending Relational Database Systems to Automatically Enforce Privacy Policies. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, Karl Aberer, Michael J. Franklin, and Shojiro Nishio (Eds.). IEEE Computer Society, 1013–1022. https://doi.org/10.1109/ICDE.2005.64

[3] Amazon Web Services Glue 2024-11-21. https://aws.amazon.com/glue/.

[4] Amazon Web Services, Lakeformation 2024-11-21. https://aws.amazon.com/lakeformation/.

[5] Anjali, Tyler Caraza-Harter, and Michael M. Swift. 2020. Blending containers and virtual machines: a study of firecracker and gVisor. In *VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event [Lausanne, Switzerland], March 17, 2020*, Santosh Nagarakatte, Andrew Baumann, and Baris Kasikci (Eds.). ACM, 101–113. https://doi.org/10.1145/3381052.3381315

[6] Apache Iceberg 2024-11-21. https://iceberg.apache.org/.

[7] Apache Livy (Incubating) 2025-03-01. https://livy.apache.org/.

[8] Apache Ranger 2024-11-25. https://ranger.apache.org/.

[9] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[10] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1383–1394. https://doi.org/10.1145/2723372.2742797

[11] Big Book of Data Engineering 2024. https://www.databricks.com/resources/ebook/big-book-of-data-engineering?itm_data=product-pc-data-intelligence-platform/.

[12] Databricks Connect 2024-12-03. https://docs.databricks.com/en/dev-tools/databricks-connect/python/index.html.

[13] Google Protocol Buffers 2024-12-03. https://protobuf.dev/.

[14] gRPC Framework 2024-12-03. https://grpc.io/.

[15] Vincent C. Hu, D. Richard Kuhn, and David F. Ferraiolo. 2015. Attribute-Based Access Control. *Computer* 48, 2 (2015), 85–88. https://doi.org/10.1109/MC.2015.33

[16] Justin J. Levandoski, Garrett Casto, Mingge Deng, Rushabh Desai, Pavan Edara, Thibaud Hottelier, Amir Hormati, Anoop Johnson, Jeff Johnson, Dawid Kurzyniec, Sam McVeety, Prem Ramanathan, Gaurav Saxena, Vidya Shanmugan, and Yuri Volobuev. 2024. BigLake: BigQuery's Evolution toward a Multi-Cloud Lakehouse. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 334–346. https://doi.org/10.1145/3626246.3653388

[17] Linux Seccomp 2024-11-25. https://lwn.net/Articles/656307/.

[18] Jacques Nadeau. 2022. Substrait: Rethinking DBMS Composability. In *1st International Workshop on Composable Data Management Systems, CDMS@VLDB 2022, Sydney, Australia, September 9, 2022*, Satyanarayana R. Valluri and Mohamed Zaït (Eds.). https://cdmsworkshop.github.io/2022/Proceedings/Keynotes/Abstract_JacquesNadeau.pdf

[19] Andrei Paduroiu, Sungheun Wi, Yan Yan, Roni Burd, Ruhollah A Farchtchi, and Giovanni Matteo Fumarola. 2024. Membrane - Safe and Performant Data Access Controls in Apache Spark in the Presence of Imperative Code. *Proc. VLDB Endow.* 17, 12 (2024), 3813–3826. https://www.vldb.org/pvldb/vol17/p3813-paduroiu.pdf

[20] Karla Saur, Tara Mirmira, Konstantinos Karanasos, and Jesús Camacho-Rodríguez. 2022. Containerized Execution of UDFs: An Experimental Evaluation. *Proc. VLDB Endow.* 15, 11 (2022), 3158–3171. https://doi.org/10.14778/3551793.3551860

[21] Xinli Shang, Pavi Subenderan, Mohammad Islam, Jianchun Xu, Jiashen Zhang, Nimish Gupta, and Ajit Panda. 2022. One Stone, Three Birds: Finer-Grained Encryption with Apache Parquet @ Large Scale. In *IEEE International Conference on Big Data, Big Data 2022, Osaka, Japan, December 17-20, 2022*, Shusaku Tsumoto, Yukio Ohsawa, Lei Chen, Dirk Van den Poel, Xiaohua Hu, Yoichi Motomura, Takuya Takagi, Lingfei Wu, Ying Xie, Akihiro Abe, and Vijay Raghavan (Eds.). IEEE, 5802–5811. https://doi.org/10.1109/BIGDATA55660.2022.10020987

[22] Spark Connect C# Client 2024-11-24. https://github.com/mdrakiburrahman/spark-connect-csharp.

[23] Spark Connect Go Client 2024-11-24. https://github.com/apache/spark-connect-go.

[24] Spark Connect .NET Client 2024-11-24. https://github.com/GoEddie/spark-connect-dotnet.

[25] Spark Connect Rust Client 2024-12-03. https://github.com/sjrusso8/spark-connect-rs/.

[26] Spark Connect SPIP 2023. https://issues.apache.org/jira/browse/SPARK-39375.

[27] Michael Stonebraker and Eugene Wong. 1974. Access control in a relational data base management system by query modification. In *Proceedings of the 1974 ACM Annual Conference, San Diego, California, USA, November 1974, Volume 1*, Roger C. Brown and Donald E. Glaze (Eds.). ACM, 180–186. https://doi.org/10.1145/800182.810400

[28] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras (Eds.). IEEE Computer Society, 996–1005. https://doi.org/10.1109/ICDE.2010.5447738

[29] Unity Catalog 2024-11-21. https://www.unitycatalog.io/.

[30] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[31] Maryann Xue, Yingyi Bu, Abhishek Somani, Wenchen Fan, Ziqi Liu, Steven Chen, Herman Van Hovell, Bart Samwel, Mostafa Mokhtar, Rk Korlapati, Andy Lam, Yunxiao Ma, Vuk Ercegovac, Jiexing Li, Alexander Behm, Yuanjian Li, Xiao Li, Sriram Krishnamurthy, Amit Shukla, Michalis Petropoulos, Sameer Paranjpye, Reynold Xin, and Matei Zaharia. 2024. Adaptive and Robust Query Execution for Lakehouses At Scale. *Proc. VLDB Endow.* 17, 12 (2024), 3947–3959. https://www.vldb.org/pvldb/vol17/p3947-bu.pdf

[32] Tao Xue, Yu Wen, Bo Luo, Gang Li, Yingjiu Li, Boyang Zhang, Yang Zheng, Yanfei Hu, and Dan Meng. 2023. SparkAC: Fine-Grained Access Control in Spark for Secure Data Sharing and Analytics. *IEEE Trans. Dependable Secur. Comput.* 20, 2 (2023), 1104–1123. https://doi.org/10.1109/TDSC.2022.3149544

[33] Tao Xue, Yu Wen, Bo Luo, Boyang Zhang, Yang Zheng, Yanfei Hu, Yingjiu Li, Gang Li, and Dan Meng. 2020. GuardSpark++: Fine-Grained Purpose-Aware Access Control for Secure Data Sharing and Analysis in Spark. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*. ACM, 582–596. https://doi.org/10.1145/3427228.3427640

[34] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

[35] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf