Blink Twice - Automatic Workload Pinning and Regression Detection for Versionless Apache Spark using Retries.

Justin Breese
Vijayan Prabhakaran
Martin Grund
Stefania Leone
Amit Shukla
Databricks
San Francisco, CA, USA

Michael Armbrust
Reynold Xin
Matei Zaharia
Lennart Kats
Sung Chiu
Tatiana Romanova
Databricks
San Francisco, CA, USA

Philip Nord
Mitchell Webster
Chris Munson
Bo Pang
David Ma
Databricks
San Francisco, CA, USA

Abstract

For many users of Apache Spark, managing Spark version upgrades is a significant interruption that typically involves a time-intensive code migration. This is mainly because in Spark, there is no clear separation between the application code and the engine code, making it hard to manage them independently (dependency clashes, use of internal APIs). In Databricks' Serverless Spark offering, we introduced *Versionless Spark* where we leverage Spark Connect to fully decouple the client application from the Spark engine which allows us to seamlessly upgrade Spark engine versions. In this paper, we show how our infrastructure built around Spark Connect automatically upgrades and remediates failures in automated Spark workloads without any interruption. Using Versionless Spark, Databricks users' Spark workloads run indefinitely, and always on the latest version based on a fully managed experience while retaining nearly all of the programmability of Apache Spark.

CCS Concepts

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

Keywords

Big Data, Apache Spark, Upgrade, Versionless, Machine Learning, Release Remediation, Pinning

ACM Reference Format:
Justin Breese, Vijayan Prabhakaran, Martin Grund, Stefania Leone, Amit Shukla, Michael Armbrust, Reynold Xin, Matei Zaharia, Lennart Kats, Sung Chiu, Tatiana Romanova, Philip Nord, Mitchell Webster, Chris Munson, Bo Pang, and David Ma. 2025. Blink Twice - Automatic Workload Pinning and Regression Detection for Versionless Apache Spark using Retries.. 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, 4 pages. https://doi.org/10.1145/3722212.3725084

SIGMOD-Companion '25, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3725084

1 Introduction

Today, upgrading Spark versions typically involves significant effort, with unclear investment requirements, including trial and error. This is mainly because in Spark there is no clear separation between the application code and the engine code. Apart from changes in the public API, any Spark internal changes may affect user workloads as users may rely on Spark internals: bug fixes in the Spark engine, changes to internal APIs, library upgrades, or language upgrades may affect customer workloads. As a result, Spark users are often reluctant to upgrade. The downside is that performance improvements, bug fixes, and new features take significantly longer to adopt, preventing customers from quickly benefiting from these improvements. In addition, it increases engineering complexity to manage a large number of different Apache Spark versions.

For Databricks serverless jobs and notebooks, we fundamentally transformed and simplified the user experience when using Apache Spark. We shifted user focus from managing Spark runtime versions to managing the stable API that they integrate with - we created client-versioned workloads with a versionless Spark server.

Decoupling the client from the Spark engine using Spark Connect has enabled Databricks to automatically upgrade the Spark server, providing users faster access to the latest features while minimizing disruptions from both intentional and unintentional breaking changes, all without compromising workload compatibility and with zero code changes needed from the user. This approach also offers significant benefits to Databricks by streamlining the release process, consolidating usage onto fewer server versions, and reducing engineering overhead from needing to backport changes.

In this demonstration, we will first briefly introduce the architectural foundation of versionless Spark, leveraging Databricks' multi-user Spark compute and Spark Connect, followed by describing in more detail how we manage seamless upgrades for our customers, and finally talk about what the user experience is.

2 Databricks Multi-User Apache Spark clusters

In Databricks, users run their workload on a standard, multi-user Spark cluster. Standard clusters provide a fully-secure multi-user capable environment that utilizes the power of Apache Spark, while offering full user ioslation, namely, client isolation and user code isolation. Client isolation is achieved using Spark Connect and user

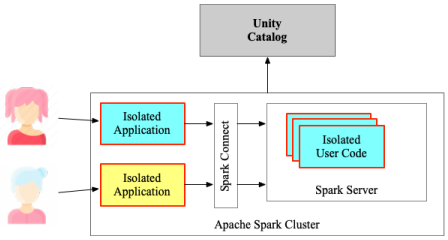

Figure 1: Fully Governed Multi-user Compute in Databricks

code isolation is achieved using Databricks' proprietary container-based sandboxing that is integrated with the Databricks cluster manager, as outlined in Fig. 1

Standard clusters are used for scheduled and ad hoc job workloads. In contrast to other vendors, the multi-user capabilities of standard clusters 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.

Databricks users 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 by any user. This complete decoupling of the application from the engine allows a Databricks standard cluster to process data in a privileged fashion and to execute queries using fine-grained access control dynamic views, row filters, column masking 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.

2.1 Client Isolation using Spark Connect

When running data and AI workloads in Apache Spark, the application code (e.g., the notebook) traditionally shares the same JVM with the rest of the engine. 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 [4], DataFrame, and Dataset [1], 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 [1, 3].

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

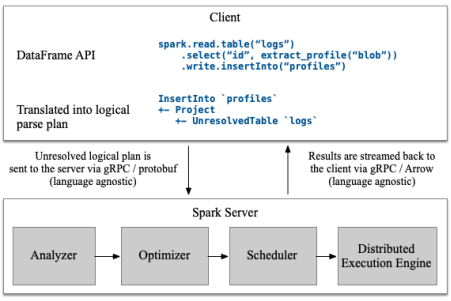

Figure 2: Spark Connect Flow

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 [2]. 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 2: 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 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.

3 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 mentioned Standard cluster architecture, providing full multi-user capabilities and allowing us

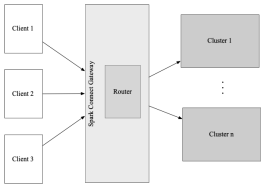

Figure 3: Basic Serverless Spark Architecture with a Workspace-Wide Endpoint

to share resources between users in the same organization securely and more efficiently.

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.

Figure 3 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 are connected to an 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.

Security There are multiple layers of isolation in the security model for serverless Spark. Each workload runs in an isolated, unprivileged container with dedicated local and attached disks, which are temporary, encrypted at rest, and automatically wiped after use. Compute is dedicated per customer and has no privileges or credentials to other systems. Finally, each workload is on an isolated logical network with no public IP addresses or ingress allowed from other workloads. Traffic is through the cloud provider's global network (not public Internet) and uses TLS 1.2+.

4 Versionless Spark

In Databricks' serverless Spark offering, we explicitly leverage the backward compatible properties of the Spark Connect protocol to provide a fully versionless engine experience to our customers. Customers and users do not have to choose a particular version of the Spark engine that runs their workload. Instead, users choose the API version they want to integrate their workloads with, and Databricks manages the engine version independently and maintains compatibility.

As mentioned in Section 3, different clients are connected to a central gateway and are then forwarded to backend compute resources. The API of the client is defined as part of the client environment.

4.1 Client Environment

The client environment manages the minimal set of dependencies our users need to run their workloads against our serverless Spark infrastructure. For Python workloads, it includes the API version of the Spark Connect client, the Python version, and additional dependencies included for convenience. We offer several versions

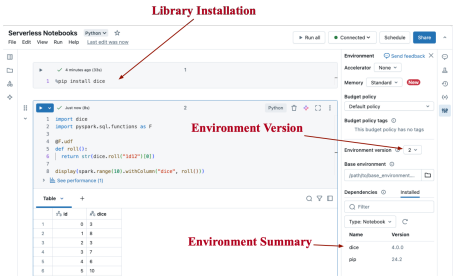

Figure 4: Serverless Notebook Environments

of client environments, called environment versions, each with different versions of the aforementioned minimal set of dependencies. In their workloads, users are free to install additional dependencies at runtime.

Whereas in traditional notebook environments it would be enough to only manage dependencies on the client side, this is not the case for Apache Spark. In Apache Spark, it is very common to extend the functionality of Spark using user code in the form of user-defined functions (UDF). UDFs are not only used for scalar transformations of individual values, but user code can be used to implement custom data sources in Python, write handler functions that are run during streaming ingestion upon batch completion, etc. To support the seamless execution of this user defined code, we are replicating the client environment dynamically to the backend cluster to execute the user code including all of its dependencies.

Figure 4 shows how users can dynamically install and manage workload dependencies in their notebooks and how to choose an environment version. For users submitting their workloads as scheduled jobs in Databricks, they can either manage the dependencies similarly in a notebook or manage them as part of the job definition.

4.2 Server Runtime

Because the Spark Connect protocol is designed to support older clients to seamlessly connect and the execution of user code being hermetically decoupled from the engine, the actual backend Apache Spark engine can be upgraded independently of the user workload. This ensures that applications automatically receive performance improvements and bug fixes without requiring any code changes.

5 Automatic Workload Remediation

Providing security, reliability, and performance are the most critical requirements for our customers when scheduling automated workloads on Databricks. Due to the tight integration between the user's application code and Apache Spark, upgrading from one version of Apache Spark to the other has always been associated with uncertainty and additional work. For workloads using Databricks serverless Spark, Databricks completely manages the upgrade experience of the Apache Spark workloads.

For every automated execution of the workload, the job runs through the abbreviated activity diagram as shown in Figure 5. For jobs scheduled in Databricks, we use a so-called workload fingerprint to identify repeated executions of the same workload based on a set of properties. This allows us to uniquely identify workloads, independently of them being scheduled in Databricks directly or using an external orchestrator. In addition, as part of

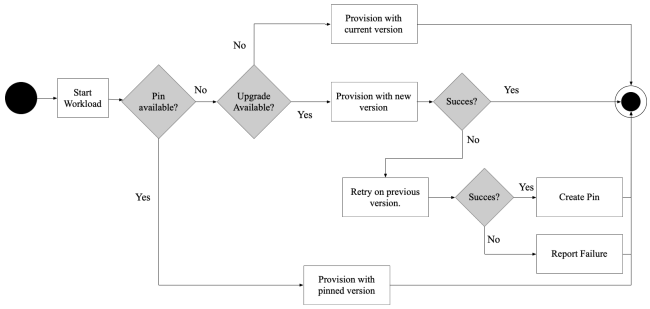

Figure 5: Automated Workload Pinning in Databricks

the health-mediated release of Databricks runtime versions, we operate what is called a pinning service. The role of the pinning service is to track workload failures that are likely to correlate with an upgrade of the Databricks runtime version. The pinning service keeps track of the workload fingerprint, the last known successful engine version, and additional workload metadata and metrics.

At the beginning of each job, the workload scheduler identifies if the pinning service has an entry for this particular workload fingerprint. If so, it will provision the workload using the last known good version. If no pinning entry is available, the process will check if a new engine version is available and whether an upgrade needs to be applied. If no upgrade is available, the execution will continue as normal. If an upgrade is available, the workflow will start provisioning resources using the new engine version. Upon success, the workload finishes normally. If the workload fails due to an error, we use a specially trained machine learning model to classify whether the error is a user error, infrastructure error, or system error. User errors will fail the job and finish the workflow. Infrastructure errors are retried on the same engine version, whereas system errors are then retried on the previously known good Databricks runtime version. If the new run, on the previous engine version for the system error, is successful, we then create a pinning entry and trigger an additional process to manage this pin. If the run is not successful, we assume that it is a user error and report the failure. The pin entry now captures the correlation signal of the same workload acting differently in two different versions of the engine.

In addition to trivial pass-fail signals, our system is able to collect additional signals, to detect performance regressions, such as workload (or task) duration and resource consumption (e.g. memory, CPU, disk) to decide whether a workload needs to be pinned based on historical runs. Some examples of these metrics include *total task time, total data read, total data written.* Integrating these additional signals is crucial for the success of our system, as the overall surface of data management systems is very large.

As briefly mentioned, creating a pinning entry triggers an additional process. First, we use a machine learning model to identify which engineering team is most suitable to do the first-level triage of the ticket based on the failure metadata such as CPU usage, memory consumption, stack traces, and server logs. In addition, the triage process starts additional data collection jobs to enrich the workload metadata with system-level observations of the workload in the Databricks environment to facilitate faster remediation.

After the pinning tickets are triaged, there are two possible situations: The ticket is a false positive, and resolving it will automatically unpin the job for the next run. If the pinning ticket is a valid bug or regression, the engineering team can associate a fix release version with the ticket, so when the new version is deployed, the workload is automatically unpinned. In addition to reacting to individual occurrence of workload pins, we use the signal to control the overall roll-out of a new server version into our fleet. Based on the large amount of data available, we make high-fidelity decisions whether to pause, continue, or slow down the roll-out of the new version.

6 Demo

For the demonstration, the audience will see a job (with retries enabled) that initially runs on an old client and an old server version - this run succeeds. The job is then triggered again and picks up a newer server version that contains a bug, causing the run to fail. The retry attempt starts, consults the pinning service, and re-runs the job on the previous (working) server version, which succeeds. We'll then unpin the workload from the pinning service, allowing the next run to use the latest server version - now fixed - which also includes a performance improvement. Finally, the job will be updated to use a newer client version to take advantage of a new user-facing API.

7 Conclusions and Outlook

Automatic workload remediation using high-fidelity workload metadata and pinning has been proven to be very successful in efficiently rolling out new Databricks runtime versions to our serverless Spark offering without forcing our customers to control the upgrade process. Since the general availability of Databricks Serverless Spark product, we have run hundreds of millions of workloads and seamlessly upgraded the engine without customer impact. The benefit for our customers is obvious: The can immediately leverage new features, stability, and performance improvements without having to plan for or manage any upgrade. Our engineering team has been able to resolve bugs faster and with more signal due to clear before and after evaluation. In addition, there is an additional engineering benefit for Databricks by reducing the number of active runtime versions in our fleet, and therefore achieve a higher overall engineering velocity.

References

[1] 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. doi:10.1145/2723372.2742797

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

[3] 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

[4] 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