A Decade of Apache Spark Structured Streaming: How We Evolved The Architecture To Meet Real-World Needs

Anish Shrigondekar
Databricks Inc.

Boyang Jerry Peng
Databricks Inc.

Siying Dong
Databricks Inc.

Jungtaek Lim
Databricks Inc.

Huanli Wang
Databricks Inc.

Bo Gao
Databricks Inc.

Eric Marnadi
Databricks Inc.

Jason Teoh
Databricks Inc.

B. Micheal Okutubo
Databricks Inc.

Livia Zhu
Databricks Inc.

Yuchen Liu
Databricks Inc.

Chloe Xia
Databricks Inc.

Abhay Bothra
Databricks Inc.

Sagar Mittal
Databricks Inc.

Karthik Ramasamy
Databricks Inc.

Michael Armbrust
Databricks Inc.

Fatih Emekci
Databricks Inc.

Shan Shan Huang
Databricks Inc.

Shrikanth Shankar
Databricks Inc.

Indrajit Roy
Databricks Inc.

ABSTRACT

Apache Spark Structured Streaming is one of the most widely used streaming platforms in the world. It currently serves millions of weekly queries across thousands of Databricks customers. This paper describes the challenges we faced and the solutions we developed while enhancing Structured Streaming to be enterprise-ready.

Structured Streaming embraces the microbatch architecture, ensuring incremental processing via a series of discrete batch computations. We discuss four noteworthy challenges and solutions in the context of Spark and the microbatch architecture. First, to address the problem of under-utilized task slots—often due to planning time and straggler tasks—we introduced microbatch pipelining, a novel extension that safely incorporates concurrency in this architecture. It improved resource utilization and throughput by almost 3x. Second, many upstream sources such as Amazon Kinesis and Google Pub/Sub did not provide semantics that aligned well with the notion of discrete microbatches. We used techniques such as async pre-fetching, deduplication management, and decoupled metadata tracking to overcome this issue and were able to support a wide variety of streaming sources. Third, the previous version of state management was opaque and inflexible, making it difficult to express complex stateful computations and debug state updates. We evolved our state management offering by adding a new flexible, arbitrary stateful operator called transformWithState and also provided fine-grained visibility into the recorded state for enhanced debuggability and observability. Finally, to enforce fine-grained access control via a sandboxed environment, we had to address novel challenges unique to long-running, stateful streaming queries, specifically around persistent connection management, cross-boundary state access, and credential lifecycle.

PVLDB Reference Format:
Anish Shrigondekar, Boyang Jerry Peng, Siying Dong, Jungtaek Lim, Huanli Wang, Bo Gao, Eric Marnadi, Jason Teoh, B. Micheal Okutubo, Livia Zhu, Yuchen Liu, Chloe Xia, Abhay Bothra, Sagar Mittal, Karthik Ramasamy, Michael Armbrust, Fatih Emekci, Shan Shan Huang, Shrikanth Shankar, and Indrajit Roy. A Decade of Apache Spark Structured Streaming: How We Evolved The Architecture To Meet Real-World Needs. PVLDB, 19(12): 4156 - 4168, 2026.
doi:10.14778/3827998.3828023

1 INTRODUCTION

Apache Spark Structured Streaming [11] has become an important component of modern data engineering platforms. Its microbatch execution model showed that streaming and batch processing can share a single semantic model. Since its introduction over eight years ago, the framework has moved beyond a novel unification of APIs to become a widely used stream processing engine in industry. Its adoption has accelerated by orders of magnitude in the last five years alone, currently powering over 8 million weekly production jobs across thousands of Databricks customers. Operating at this scale required solving a series of challenges that arose from maintaining long-running streaming workloads in production.

Structured Streaming is built on a microbatch architecture: it processes unbounded streams by breaking them into a series of discrete and deterministic batch computations. While the microbatch design provides scalability, multi-tenant execution, strong guarantees around fault tolerance and exactly-once semantics, it also introduces operational challenges that we realized as we onboarded more and more customer workloads. Our experience scaling Structured Streaming revealed four important gaps between our initial implementation and what customers wanted.

Taming the utilization problem. While Structured Streaming provides high throughput, in many workloads we saw resource utilization sometimes even below 20%. The micro-batch execution processes batches sequentially: plan, execute, commit, then repeat. This architecture simplifies reasoning about fault tolerance and

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828023

correctness but can suffer from low CPU utilization due to stragglers, I/O wait times, and query planning overhead. Low utilization impacts cost and performance, both of which our customers care about. This observation led us to enhance the current architecture with *microbatch pipelining*, a concurrency model that allows multiple batches to execute simultaneously when safe to do so. Microbatch pipelining has resulted in higher utilization and up to 3x improvement in throughput for many workloads, without changing a single line of user code (discussed in § 3).

The connector challenge. Our second major challenge emerged when customers wanted to read from sources like AWS Kinesis and Google Pub/Sub. These systems are different from sources like Kafka—wherein they don't have the notion of "the last offset" that can be queried upfront [7]. Queueing systems such as Google Pub/Sub only guarantee at-least-once message delivery [22] with acknowledgements required for processed messages. But Spark's micro-batch model requires knowing exactly what data to process before a batch begins and assumes that the source can be replayed from a given offset. To address the issue of defining batch boundaries when the source cannot report where the data ends, we developed a variety of techniques such as async pre-fetching, deduplication management, and decoupled metadata tracking that bridge the semantic gap between Spark's discrete batches and different data sources' APIs. (discussed in § 4)

Improving state management. We evolved the state management subsystem to address historical challenges around performance and flexibility (discussed in § 5). First, we evolved the underlying storage layer to improve checkpointing performance and data validation. Second, we introduced a new arbitrary state API that replaces rigid, monolithic state objects with flexible, composite data structures and native support for automated expiration. Third, we created a state store reader that exposes the internal state as a queryable data source, enabling users to inspect and debug streaming state directly using standard batch queries.

The governance gap. Another challenge came from a different direction: security and multi-tenancy. As Structured Streaming grew in adoption, enterprise customers increasingly wanted to run untrusted code—user-defined functions, third-party libraries, etc.—in their streaming pipelines. But traditional Spark runs user code in the same process as the engine, creating potential security vulnerabilities. We needed to isolate untrusted code without breaking streaming's operational model. Our Lakeguard [24] architecture applies zero-trust principles through Spark Connect (client-server separation) and container sandboxing. But streaming queries are fundamentally different from batch jobs: they run for days or weeks, maintain persistent state, and require long-lived credentials. We had to solve new problems around connection persistence over extended periods, managing state access across security boundaries, and handling credential lifecycle in sandboxed environments. The result is a system where customers can safely run arbitrary streaming code in multi-tenant environments. See § 6 for details.

The rest of the paper describes the changes we made to support enterprise Structured Streaming workloads and the lessons we learned from operating the system in production.

These lessons are relevant beyond Databricks because they show how the Bulk Synchronous Processing (BSP) [37] model can be extended for streaming workloads. Microbatch pipelining shows that parts of consecutive batches can safely overlap while preserving fault tolerance. The Lakeguard integration shows how zero-trust isolation can be applied to long-running, stateful streaming workloads, not only to short-lived batch jobs. The connector work shows how a microbatch engine can support sources that do not expose Kafka-style offsets, such as Pub/Sub.

2 BACKGROUND

2.1 Apache Spark Structured Streaming

Structured Streaming simplifies incremental processing through a declarative API, allowing users to express streaming computations with familiar Spark SQL and DataFrame semantics [12, 40]. Listing 1 provides a sense of the Structured Streaming API. It demonstrates a query where a streaming source is joined to a static table and aggregated over a time window. [1]

```
agg_df = spark.sql("""
  SELECT
    window(s.timestamp, '5 minutes') as time_window,
    t.city, count(*) as event_count
  FROM STREAM read_kafka(
    bootstrapServers => 'kafka:9092',
    subscribe => 'events') s
  WATERMARK s.timestamp DELAY OF INTERVAL 30 SECONDS
  JOIN user_info t
  ON get_json_object(CAST(s.value AS STRING),
                     '$.user_id') = t.user_id
  GROUP BY
    window(s.timestamp, '5 minutes'), t.city """)
agg_df.writeStream.format("delta").outputMode("append") \
      .option("checkpointLocation", "/tmp/chkpt") \
      .toTable("event_by_city")
```

Listing 1: An example Structured Streaming query.

The programming model combines three components: replayable input sources (e.g., Apache Kafka [28]), incremental processing that updates a logical result table, and idempotent or transactional sinks that durably write results and handle micro-batch rewrites. Both sources and sinks are extensible through connectors. Structured Streaming reuses Spark SQL's Catalyst optimizer and runtime code generator. This paper focuses on micro-batch mode, which represents the vast majority of real-world deployments; continuous processing and Spark 4.1 real-time mode are out of scope.

2.2 The micro-batch architecture

Structured Streaming's micro-batch architecture follows the Bulk Synchronous Processing (BSP [37]) paradigm, whose widely used internet-scale example is MapReduce [19]. The system reads a short input interval, processes it through a directed acyclic graph of operators, and then advances to the next batch. Batch boundaries give the engine natural points for scheduling, resource reallocation, and checkpointing; recovery can simply re-execute the failed micro-batch in parallel across nodes.

Low-latency systems such as Apache Flink [14] and Apache Storm [36] instead use continuous operators: long-lived DAG nodes

[1] Reading a Kafka source using the read_kafka() table-valued function (TVF) is specific to the Databricks environment. In open-source Apache Spark, equivalent functionality is achieved by defining a temporary view for the streaming source and referencing that view in the SQL query or by writing the equivalent logic directly with the DataFrame API.

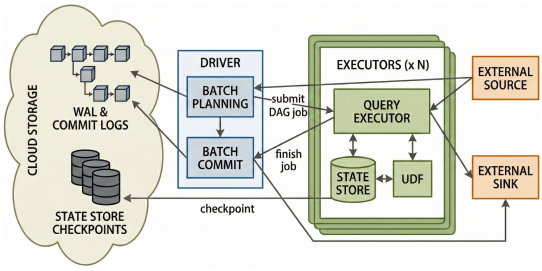


Figure 1: Execution Lifecycle of a Micro-Batch.

through which input data flows without BSP-style synchronization barriers. This model is well suited to low latency, but recovery requires distributed checkpointing and previous work has shown that throughput can be worse than micro-batch architectures [27].

2.3 Execution lifecycle

Historically, micro-batches execute strictly sequentially through the following lifecycle (Figure 1):

(1) **Planning & offset discovery**: The Spark driver queries the input source to determine the exact start and end "offsets" (data boundaries) for the upcoming batch.
(2) **Journaling**: These planned offsets are durably written to a Write-Ahead Log (WAL, called Offset Log in Spark).
(3) **Execution**: The engine builds a Directed Acyclic Graph (DAG) query plan. Based on data dependencies, execution is divided into stages; each stage is further split into tasks that run in parallel across cluster nodes. For Listing 1, the stages may read Kafka, read user_info, join, pre-aggregate, and then finalize the aggregation and write to the sink.
(4) **Watermark propagation**: The engine calculates and updates the global watermark to determine which state is eligible for eviction in future batches.
(5) **Commit**: A final entry is written to the Commit Log to signal the batch is complete, and the cycle repeats.

Sinks (e.g., Delta Lake) often adopt a two-phase write protocol, where parallelizable data writes (e.g., writing to Parquet files) execute in Step 3, and the final commit (e.g., updating table metadata) occurs in Step 5.

Upon batch completion, StreamingQueryListener callbacks receive monitoring events. This lifecycle is restartable: after a crash, the engine compares the offset and commit logs and replays any batch present in the offset log but absent from the commit log.

2.4 State management architecture

A query is stateless if it processes each record or batch independently, and stateful if it must retain prior data for windowed aggregations, session tracking, or stream-stream joins. Structured Streaming uses a two-tier architecture: mutable state stays local to compute nodes for fast access, while snapshots are checkpointed to durable remote storage at micro-batch boundaries.

Beyond built-in aggregations, Spark provides *Arbitrary Stateful Processing* through operators such as flatMapGroupsWithState, where user code reads and updates per-key state with timeout control. For example, a user can group activities into security related monitoring sessions and automatically expire each session after inactivity.

2.5 The foreachBatch operator

For writing logic beyond standard sinks, Structured Streaming provides foreachBatch, which applies arbitrary user logic to each micro-batch output. It is used by more than 50% of streaming queries in Databricks. The operator executes synchronously, enabling writes to multiple downstream locations or systems without native streaming support.

2.6 Databricks Lakeguard

Historically, Spark ran user code such as UDFs in the same process as the engine driver. This created a governance gap: malicious or poorly written code could bypass security layers and access engine memory or data files.

Lakeguard [24] fixes this by combining two key architectural principles: client decoupling and containerized isolation. Spark Connect separates clients from the server: clients submit logical plans that the server optimizes and executes. Lakeguard then intercepts user-defined functions (UDFs) and runs them in secure, sandboxed containers outside the engine JVM.

3 MICROBATCH PIPELINING

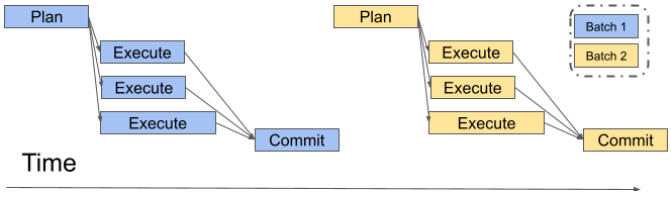
Structured Streaming executes micro-batches sequentially: planning, execution, and commits are strictly ordered to simplify correctness reasoning. However, this sequential workflow often yields low resource utilization, under 20% in some benchmarks. This inefficiency arises as nodes remain idle during batch planning, wait for the completion of high-latency straggler tasks, or stall when individual stages provide insufficient parallelism to saturate the cluster's compute capacity. Although multi-tenancy offers a way to mitigate low utilization by co-locating multiple queries on the same cluster, its effectiveness is often hampered by the difficulty in finding perfectly co-locatable workloads and the challenge of tuning for stable latency in this mode, thereby creating usability challenges. We introduce micro-batch pipelining, a technique that parallelizes the stateless phases of future batches with the stateful and commit phases of current batches, effectively reclaiming idle cluster resources without violating streaming semantics. Figure 2 shows how pipelined and non-pipelined executions differ.

3.1 What can (and cannot) be pipelined

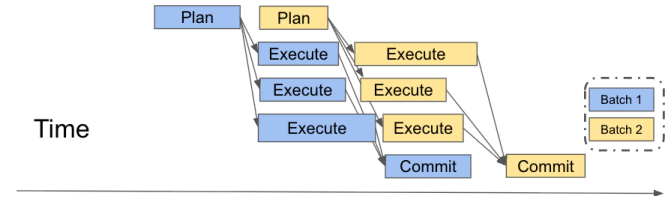
The core insight for micro-batch pipelining is that while batches are logically sequential, their physical execution phases do not all share the same dependencies. By analyzing the lifecycle of consecutive batches (N and $N+1$), we can categorize operations into those that can execute concurrently versus those that must be serialized.

Parallelizable operations:

- **Stateless stages:** The stateless transformations (e.g., map, filter, projection) of batch $N+1$ have no data dependency on batch N, except in cyclic pipelines where an output



(a) Single-Stage: Sequential



(b) Single-Stage: Pipelined

Figure 2: With microbatch pipelining, the second batch starts processing sooner, resulting in higher throughput.

serves as an input. These stages can execute freely in parallel with any phase of the previous batch, effectively utilizing idle cluster resources.

- **Data staging:** For sinks that support two-phase writes, the parallel data-writing phase of batch $N+1$ can proceed concurrently with the commit phase of batch N.

Non-parallelizable operations:

- **Stateful stages:** Any stage involving stateful operators (e.g., aggregations, stream-stream joins) must be strictly serialized. The stateful stage of batch $N+1$ cannot begin until the stateful stage of batch N completes, as they share the same underlying state stores.
- **Log maintenance:** Writes to the Offset Log and Commit Log must occur in strict increasing order to ensure deterministic failure recovery and exactly-once semantics.
- **Sink commits:** The final commit operation for transactional sinks must be serialized to prevent readers from seeing results out of order.
- **Metric reporting:** Progress events must be reported in batch order to preserve the logical timeline for monitoring systems, even if physical execution overlaps.

3.2 Challenges in concurrent stream execution

Moving from sequential to pipelined execution requires changes across several parts of the engine, beyond adding threads:

Execution context isolation: The existing engine relies on a singleton execution context where mutable state—such as start/end offsets, watermarks, and metrics—is shared globally. Concurrency turns this into a race condition. To support pipelining, we must decouple this global state into isolated, per-batch contexts. Also, state propagation (e.g., batch N's end offsets becoming batch $N+1$'s start offsets) must transition from implicit ordering to explicit, thread-safe transfer between concurrently executing objects.

The scheduling problem: Determining the optimal start time for batch $N+1$ is a complex optimization problem. Starting too early causes the batch to stall on dependencies (e.g., state locks), holding resources without making progress. Starting too late leaves the cluster idle, failing to mask the latency of the previous batch. The system must implement an adaptive scheduling algorithm that uses historical measurements to predict when the stateless prefix of batch $N+1$ will finish as the stateful stage of batch N completes.

Enforcing cross-batch dependencies: Spark's DAG Scheduler is historically designed for single-job dependencies. Pipelining requires extending the scheduler to enforce constraints across separate jobs: specifically, withholding the stateful stage of batch $N+1$ until batch N's stateful stage completes, while simultaneously allowing batch $N+1$'s stateless stages to proceed in parallel.

Serializing ordered operations: While compute tasks run concurrently, specific lifecycle operations must remain serialized to preserve correctness. Offset and commit logs must be written in strict sequence to ensure deterministic recovery. Similarly, transactional sinks (like Delta Lake) must commit in order to prevent readers from seeing results out of sequence, and metric reporting must follow batch order to avoid confusing monitoring systems.

Source and sink API limitations: Legacy connectors that assume sequential execution must be modified to support pipelining, whereas those already utilizing two-phase writes are naturally well-suited for it.

Complex failure recovery: The standard sequential recovery invariant of replaying the single batch present in the offset log but absent from the commit log breaks under concurrency. With multiple batches inflight, a failure may leave a gap of uncommitted batches (e.g., batch N is committed, but $N+1$ is not). Recovery logic must be redesigned to scan back and identify the oldest uncommitted batch among all inflight contexts, replaying the entire sequence from that point to ensure exactly-once semantics.

3.3 System design

We discuss the redesigned execution engine that enforces dependencies at the scheduler level.

3.3.1 Cross-job dependency management. A core change is the extension of the Spark DAG Scheduler to support dependencies that span distinct jobs. We identify "stateful stages" (those accessing state stores) and enforce a DAG constraint where the stateful stage of batch $N+1$ cannot launch until the stateful stage of batch N completes. The scheduler also unblocks the stateless parent stages of batch $N+1$, allowing them to execute concurrently with batch N and fill the idle gaps typical of sequential processing.

3.3.2 Adaptive feedback scheduling. We implemented an adaptive control loop to solve the optimal start-time problem. The scheduler maintains a rolling weighted average of recent stateful stage durations to predict the ideal inter-batch gap. This estimate is continuously refined by two signals:

(1) **Task launch delay:** If tasks wait for slots, the system infers it started too early (resource contention).
(2) **Early submission:** If stateless stages finish before dependencies are ready, the system infers a premature start.

An additive increase/decrease algorithm adjusts the wait time based on these signals, converging on a schedule that maximizes overlap while minimizing contention.

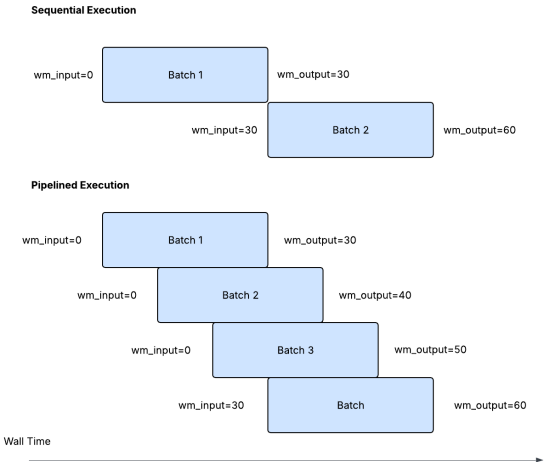


Figure 3: Advancing watermark: pipelined execution runs up to 3 batches in flight, and the watermark advances at a uniform rate relative to data processed.

3.3.3 Watermark. Each microbatch computes a new watermark after its computation completes. Subsequent batches use this watermark for operations such as state eviction and other event-time-based logic. Without pipelining, each batch inherits the updated watermark from its immediate predecessor. Pipelining breaks this strict sequential inheritance: batch $N + 1$ often begins planning before batch N has finalized its watermark update, so it may be planned with a stale watermark. This is not a correctness concern. Watermark semantics only require monotonic advancement — the rate of advancement is a performance property, not a correctness one. Per batch, watermarks advance more slowly under pipelining: with up to k batches in flight, several consecutive batches may be planned with the same watermark. In wall-clock terms, however, the global watermark tracker still folds in every completed batch's contribution serially. Because pipelining drains more batches per unit time, the tracker advances at least as fast in wall-clock time — and typically faster, since more data is ingested over the same interval. Figure 3 depicts watermark advance with and without pipelining.

3.3.4 Correctness and recovery. We preserve strict serializability for side effects by combining offset logging and sink commits with a synchronized phase protocol. For failure recovery, we address the divergence between the write-ahead log and the commit log. Unlike the sequential case, multiple batches may be inflight during a crash. We enforce an invariant where log entries are written in strict batch order, ensuring that replaying from the oldest uncommitted batch correctly restores the state, regardless of the degree of concurrency at the time of failure.

3.4 Performance evaluation

We used an ETL-style workload to evaluate the effectiveness of micro-batch pipelining. Our goal is to measure the throughput improvements. We used a sampled 1 TB dataset and ran streaming queries on a cluster with 10 worker nodes, each provisioned with 8 cores. We evaluated both stateless and stateful ETL workloads that read from a Delta table (or object store) and write results to another Delta table. Figure 4 shows the resulting throughput– the stateless workload sees up to 2.8x throughput improvements, while the stateful workload's throughput improves by almost 2.5x.

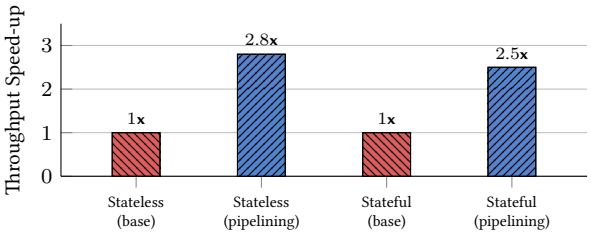


Figure 4: Microbatch pipelining speedup for ETL workloads

In another benchmark, we compared the cost and performance of ingesting 100K JSON files into a Delta table in Databricks. When microbatch pipelining is enabled, Figure 5 shows about 45% reduction in cost, due to the higher throughput.

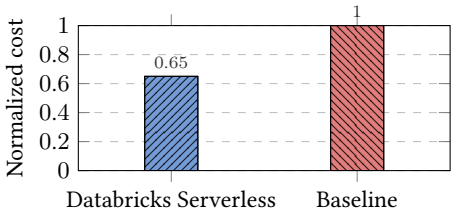


Figure 5: Cost benefit of microbatch pipelining.

Another common workload is ingestion. We created an ingestion benchmark that ingests approximately 2 TB of data by reading from Kafka and writing to Delta. In this benchmark, Structured Streaming with Microbatch Pipelining enabled was compared against EMR Flink (pre-flink 2.0) and delivered approximately 1.8× higher performance, as shown in Figure 6. Because the data is pre-generated, this benchmark inherently favors Flink by eliminating the idle time tasks spend waiting for new data. Nevertheless, by leveraging Micro-Batch Pipelining, Structured Streaming achieves comparable slot utilization to Flink, with the Databricks Photon Engine [13] driving further performance gains through enhanced processing efficiency.

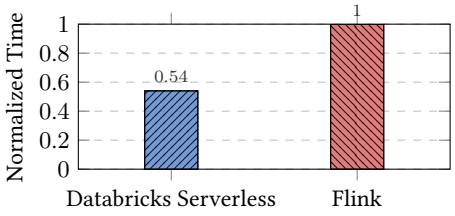


Figure 6: Ingestion time from Kafka to Delta Lake.

3.5 Real-world deployment challenges

Feature incompatibilities. Engine features designed under the assumption that batch N fully completes before batch $N + 1$ begins may encounter issues. The clearest case is Delta Lake's auto-incrementing IDENTITY column, for which reserving identifier space for an in-flight batch $N + 1$ is non-trivial; pipelining is therefore disabled when such columns are present.

Performance regressions under extreme concurrency. Under very high query concurrency, the smaller and frequent batches produced by pipelining can saturate the task scheduler, and the aggregate overhead outweighs the latency benefit. We are actively researching ways to refine the mechanism and resolve these real-world edge cases.

Implicit API ordering contracts in connectors. Prior to pipelining, sequential execution incidentally produced the invariant that a source's $\mathrm{commit(N)}$ always followed the metrics report for batch N. Although never part of the documented source API contract, certain connectors had come to depend on it, since their metrics are derived from state that $\mathrm{commit()}$ mutates. Pipelining allows batches N and $N + 1$ to overlap, letting the commit of batch N — issued from batch $N + 1$'s thread — race with or precede batch N's own metrics reporting, yielding incorrect progress. We resolved this by snapshotting source metrics before the commit log entry for batch N is written, restoring the original observable ordering irrespective of pipelining.

4 BUILDING STREAMING CONNECTORS

Structured Streaming aims to provide exactly-once processing guarantees and event-time semantics for all supported sources. To support this, Spark needs to define clear end offsets for each source for each micro-batch to determine what data to process and, crucially, what offsets to commit to guarantee no data loss or duplication upon recovery. It also requires that sources be replayable so that it can read a deterministic set of values for a given range of offsets in order to support recovery scenarios. In the following sections, we will describe how we satisfy these constraints for different classes of sources, which may not naturally align with the requirements placed by micro-batch semantics.

4.1 Sources with no end offsets

Unlike some streaming sources, Amazon Kinesis does not provide a direct API to query the *latest* sequence number upfront. This means the streaming engine cannot fetch the latest offset and record it in the streaming checkpoint, which is necessary to plan the microbatch in relation to the last committed batch. Also, Amazon Kinesis provides two consumer modes which have certain behavioral differences for clients: a polling mode which requires clients to periodically poll Amazon Kinesis for new data and an enhanced fan-out mode which is a push-based model that requires clients to register subscriptions with Amazon Kinesis and then receive push-based events.

Polling mode To achieve exactly-once processing semantics with Structured Streaming, the Databricks Kinesis connector uses an asynchronous prefetch thread on the Spark driver.

- Prefetch Cycle: This thread periodically initiates Spark jobs to parallelize data retrieval from Kinesis shards.

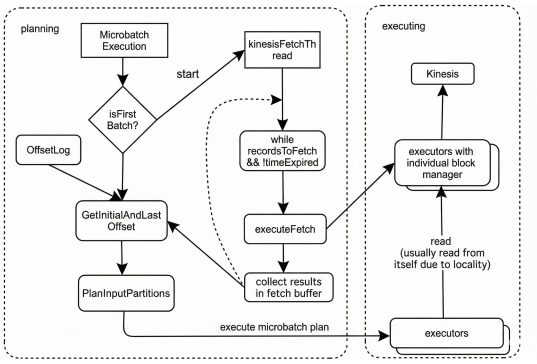


Figure 7: Kinesis Source Connector.

- Data Buffering: Metadata for retrieved records is stored in a FetchBuffer on the driver and the data itself is cached in executor memory via the Spark BlockManager.
- Batch Planning: During query planning, the latest prefetched record defines the end offset for the next micro-batch. This allows the driver to provide the executor tasks with specific block references for processing, significantly reducing latency by optimizing for spatial locality and avoiding redundant fetches during the execution phase.

Enhanced fan-out (EFO). In the enhanced fan-out mode, we similarly register subscriptions with Kinesis during each pre-fetch cycle and can fine-tune and optimize the subscription duration to align with restrictions placed by the Kinesis service. For cache misses, Spark falls back to reading until the recorded end offset in polling mode which offers more control for such scenarios and does not require us to handle subscription conflict management across multiple threads. Note that in this model, we pay the penalty of reading and recording a batch worth of data through the prefetching scheme which could incur higher latencies than stricter record-by-record processing techniques.

Resharding and fault tolerance. A critical enterprise requirement for our Kinesis source customers is also the ability to handle resharding (splitting or merging shards) without data loss or manual intervention. The connector periodically lists and updates the list of active shards and continually tracks split, coalesced and closed shards. We ensure exactly-once processing through strict checkpointing of sequence numbers. The connector tracks every record's unique sequence number within a shard to ensure strict ordering and replayability.

4.2 Queue-based sources

Queue-based sources like Google Pub/Sub pose a different set of challenges. These upstream systems don't have the notion of offsets, but rather are based on message deliveries and acknowledgements. In addition, these systems usually advertise at-least-once delivery semantics, which means that there is a non-zero chance of a message being redelivered to the client despite being acknowledged earlier. Since a large number of our customers also rely on these types of sources for various workloads, we needed to support them with Structured Streaming as well. In the following sections,

we describe how we solved micro-batching requirements for these types of upstream systems:

State store based deduplication management. One of the primary challenges with Google Cloud Pub/Sub [21] is its lack of built-in ordering and the potential for message redelivery, which can lead to duplicate records. To solve this, the connector implements a deduplication layer using a state store (backed by RocksDB) to track every record by its unique message ID. After parallel fetching, all messages are shuffled by their ID to ensure that duplicates are routed to the same state store partition for comparison. This state-based approach allows the connector to identify and filter out already-processed messages before they ever reach the final user-facing batch, ensuring strict exactly-once guarantees.

Data storage and filtering. Since the upstream system also does not offer the ability to ask for specific offsets or records, we need to store the message data on our own. We decided to store the raw records as Parquet files on object storage. However, we also need to ensure that duplicate records are not retrieved and read by the query engine. Removing duplicates from high-volume data streams is expensive if it requires rewriting entire files. We manage this using Deletion Vectors [20], which are bitmaps that indicate which specific rows in a stored Parquet file should be ignored during reading. Instead of discarding duplicates at the time of ingestion, the source writes all fetched records to Parquet files and simultaneously updates the state store. The state store then emits the row numbers and file names of any duplicates, which are compiled into a deletion vector passed to the Spark engine. This allows the system to omit duplicate rows at read-time, optimizing both storage and compute performance.

Fetch-deduplication fault tolerance. Maintaining data integrity during system failures is critical, especially when failures occur midway through a multi-step cycle involving fetching, deduplicating, and acknowledging messages. The connector addresses this by utilizing a retry mechanism and epoch-based tracking. If a failure happens after data is fetched but before it is deduplicated, the system renames those files into a "retry" directory and completes the deduplication before attempting any new fetches from Pub/Sub. This mechanism ensures that no data is lost and that messages are only acknowledged (ACKed) once they have been safely processed and recorded in the system's state.

Performance tuning. Scaling a streaming source requires balancing throughput against the latency of the microbatch thread. Common challenges include lagging fetch threads or slow deduplication due to state store overhead. To mitigate these, we also allow developers to fine-tune their usage further using parameters such as `numFetchPartitions` for parallelizing requests and `maxFetchPeriod` to optimize batch sizes. Furthermore, to combat RocksDB synchronous commit latencies, the system leverages changelog checkpointing, which helps maintain high performance even as the volume of unique message IDs in the state store grows.

4.3 File-based sources

Raw files (CSV, JSON, Parquet, etc.) on object storage are another class of upstream sources that our customers want to ingest. These files themselves don't tend to provide metadata tracking of their own and this needs to be managed within the streaming connector.

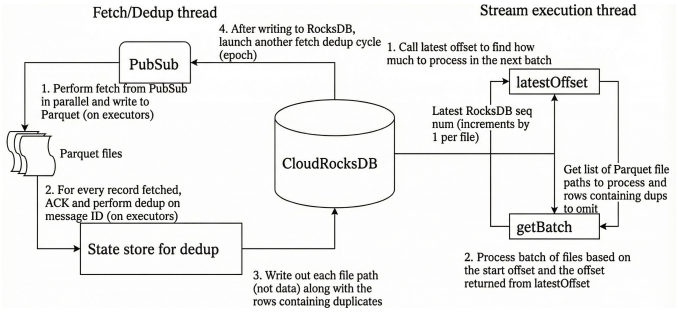


Figure 8: Pub-Sub Source Connector.

Newer lakehouse storage formats such as Delta [10] and Iceberg [31] alleviate this problem by providing replayable versioning and metadata tracking that tend to work well with micro-batch based systems. For raw files, however, we need to figure out which files are new, which have been ingested to completion, schema evolution and other factors. To manage file discovery, our connector offers two primary modes: Directory Listing, which efficiently scans the storage for new files using lexical ordering optimizations, and File Notification, which uses cloud-native messaging services (like AWS SQS or Azure Event Grid) to trigger ingestion based on file-creation events. By maintaining file-level metadata in a scalable RocksDB-backed checkpoint on the Spark driver, our connector ensures exactly-once processing guarantees even when dealing with billions of files or near real-time ingestion rates. Beyond efficient file discovery, our connector also significantly reduces operational overhead through automated schema inference and evolution. It can automatically detect the data structure of incoming files and adapt to changes—such as the addition of new columns—without requiring manual intervention or pipeline restarts. To handle data quality and schema drift, it features a rescued data column, which captures any data that does not match the expected schema (e.g., unexpected fields or mismatched data types).

5 STATE MANAGEMENT

State management preserves operator state across micro-batches, including aggregation buffers, join state, timers, and user-defined state (see § 2.4). Stateful operators are crucial for advanced streaming ETL as well as operational workloads that might need to rely on windowed aggregations, stream-stream joins or even expressing arbitrary stateful logic. We enhanced the state management subsystem around two key themes: fundamentally improving the efficiency and observability of the underlying state store, and providing a flexible, next-generation API called `transformWithState` for complex business logic. Barring some performance enhancements, all the other functional improvements and API changes listed in the sections below were done in close collaboration with the open-source [32] Apache Spark community.

End to End Latency (millisec)	P50	P95
Baseline	1716	5696
With changelog Checkpointing	652	1281
With changelog Checkpointing and micro-batch pipelining	570	1223
All above and async checkpointing	474	1144

Table 1: Performance of Windowed Count Aggregation (50K records/sec, 1200-byte keys) on a 5-node r5d.2xl Cluster (8 vCPUs, 64GB RAM per node)

5.1 Tackling inefficiency and opacity

Our previous state store implementation suffered from many operational challenges. It was inefficient because it relied on synchronous checkpointing and a rigid file format that relied on deterministic updates and atomic overwrites. During a micro-batch, the engine had to wait for the entire state to be written to cloud storage before it could proceed, causing latency spikes for queries with large state sizes. Additionally, the legacy checkpoint format was prone to overwriting risks and corruption from zombie tasks because it relied on deterministic file names and atomic overwrites, a guarantee that is not provided by all cloud providers. The system also exposed little visibility into state store contents; users could only verify state correctness indirectly through the query's output records, making debugging and unit testing nearly impossible. To address these fundamental flaws, we introduced four major architectural improvements:

Changelog and async checkpointing. To reduce and hide the "stop-the-world" latency caused by synchronous checkpointing, we transitioned to uploading incremental changelogs for all the batches. Full snapshots are generated periodically and uploads still happen asynchronously. With async checkpointing, changelog uploads and optionally local snapshot creation are run in an async future avoiding blocking the task and thereby stage execution in Spark. This allows for scheduling the next stateful stage and the tasks in this subsequent stage can execute as long as the local database version for these partitions is at the expected version, helping boost performance further. Table 1 shows latency benefits of these improvements.

For stores like RocksDB, we keep persistent updates limited to data files and disable writes to the write-ahead-log (WAL). The reason is that we only require batch-level atomicity which is captured via changelog files. Extending asynchronous snapshotting beyond task boundaries introduces two major architectural challenges. First, Spark's best-effort task affinity permits state store migrations that disrupt background snapshots (mitigated by our maintenance improvements, described shortly below). Second, background snapshotting severely exacerbates distributed race conditions inherent to speculative execution, task retries, and the re-computation of duplicated sub-plans (resolved by our lineage-based checkpointing format).

Lineage-based checkpointing format. Beyond increasing the system's robustness against asynchronous race conditions just described, strict lineage is required to prevent data corruption during non-deterministic executions. For example, under task retries or speculative execution, batch $N+1$ might build upon the state of batch N attempt N_1. If attempt N_2 subsequently executes, applying the $N+1$ changelog to N_2's state creates a divergent history. While benign for deterministic workloads, this mismatch silently corrupts queries containing non-determinism.[2] To resolve the risks, we redesigned the checkpoint format. Every checkpoint is now saved with a globally unique ID and embeds its own lineage information. This provides a strict guarantee of linear lineage, ensuring that other attempts of the same task cannot overwrite it.

This checkpointing pattern applies to both changelog files as well as full snapshots. Fundamentally, each distinct execution attempt of a task for a given batch independently generates a checkpoint with a globally unique ID. The relevant mapping for the individual state partition and its corresponding tracking files on checkpoint storage are persisted in the commit log so that they can be used to decide the lineage for future batches. Because the commit log acts as a centralized registry, the checkpoint ID recorded within it serves as the deterministic, global source of truth for state lineage. [3]

Maintenance management improvements. State store maintenance plays a key role in partition recovery, either on query restart or task node reassignment due to an executor failure or due to new node additions based on auto-scaling signals. It is important for us to ensure that full snapshots for a given partition do not lag beyond acceptable thresholds in order to ensure fast, predictable partition recovery times. Best-effort task affinity can lead to severe snapshot starvation when state stores frequently migrate across nodes. To mitigate this starvation and ensure fair processing across partitions, we have added numerous improvements to track snapshot lag and force synchronous uploads during query execution as a back-pressure mechanism to keep the lag within bounds. We also support delayed eviction of out-of-retention files to increase the success chances of uploads by trading off cloud object deletions. We also added support for performing snapshot repair using a lineage of old full snapshots when the latest snapshot fails to load. Using this approach, we can repair the existing checkpoint rather than asking users to throw away the checkpoint and perform expensive reprocessing and building the state from scratch.

State data source reader. We introduced the State Store Reader to allow users to inspect the content of the state store for query development, troubleshooting, and normal monitoring. This feature treats the streaming state store as a virtual table that can be easily queried and analyzed. Spark users can now inspect the ground truth of their state offline, specify particular batch versions or operator IDs, and debug complex streaming jobs using standard SQL—without needing the streaming query to be actively running. We also allow for even fine-grained tracking, where the user could read the state updates similar to reading a change-data feed (CDF) for the state store.

The checkpointing format helps with offline analysis. Since all the state store contents are persisted to the streaming query checkpoint, we can rebuild the state written by the operator at any given

[2]For instance, in SELECT random_key, COLLECT_LIST(payload) AS payload FROM (SELECT *, CAST(RAND()*1024 AS INT) AS random_key FROM your_table_name) GROUP BY random_key, RAND() evaluates differently across retries; applying subsequent updates to a mismatched state variant silently drops payloads.

[3]Details are available in https://issues.apache.org/jira/browse/SPARK-49374

```
// TWS makes complex state management, like TTL,
// a simple configuration
val ttlConfig = TTLConfig(Duration.ofSeconds(600))

// Define a fine-grained ValueState with automatic
// 10-minute expiration. The engine automatically
// evicts stale state without custom user logic.
sessionStateVar = getHandle.getValueState("
    sessionStateWithTTL",
  Encoders.scalaInt,
  ttlConfig)
```

Listing 2: TWS Lifecycle and TTL Configuration

batch using a combination of full snapshots and changelog files. Operator-specific metadata is stored separately that encodes information about used configurations as well as recorded schema. This metadata can be used to retrieve the contents of the state store in a human-readable format using the DataFrame APIs. Although initially designed for diagnostic troubleshooting and offline analysis, we also observed an emergent, unanticipated use case: query state migration. Users actively leverage the state store reader as an upstream input source, transforming the extracted state records and loading them as the initial state for arbitrary stateful operators (detailed in the next section).

5.2 Rethinking stateful APIs: The transform WithState operator

While the underlying state store improvements resolved operational bottlenecks, developers still faced significant API limitations. The previous generation of arbitrary stateful operators, such as mapGroupsWithState or flatMapGroupsWithState, were rigid.

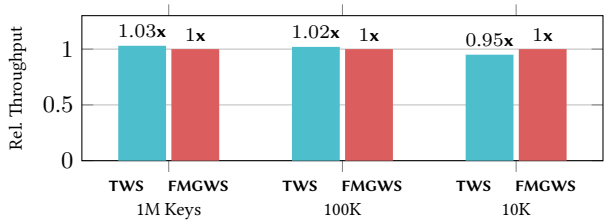
These legacy operators forced a coarse-grained, "all-or-nothing" state model. Users had to represent their entire state as a single, monolithic object. If a user only needed to update a single boolean flag or append an item to a list, the engine still had to fetch, deserialize, update, and re-serialize the entire state object. Implementing complex features like composite data structures or fine-grained Time-to-Live (TTL) eviction was manual and error-prone, requiring developers to write custom garbage collection logic inside their business functions. Also, these operators only supported the notion of timeouts which were recorded on a per-row basis and required expensive full data scans to check for expiration.

We solved this by introducing a new operator called transformWithState (TWS). Instead of a monolithic object, TWS is an interface-driven operator that supports fine-grained, composite state types such as ValueState, ListState, and MapState. It allows users to interact with state variables independently, drastically reducing serialization overhead. Furthermore, it introduces native timer management where users could register multiple unique timers per keyed state as well as automated TTL.

Previously, expiring a user session required manually tracking timestamps inside a custom state object and iterating through the entire state to prune old records on every batch. With TWS, managing the state lifecycle is trivial and handled natively by the engine on a per-state variable basis. Listing 2 shows an example TWS code.

By providing granular state primitives, transformWithState automates lifecycle management, mitigates unbounded state growth, and lets developers express complex event-driven logic directly.

Although the large majority of customer workloads using the transformWithState API are net-new, we have also seen some customers migrate their legacy flatMapGroupsWithState workloads without much effort and take advantage of the new API.



(a) Value based update benchmark in Scala across different key cardinalities.

(b) TopK benchmark in Scala (10k keys) with varying K values.

(c) List append benchmark in Scala (10k keys) with varying list sizes.

(d) Map lookup benchmark in Scala (1M keys) with varying user key cardinalities.

Figure 9: Benchmarks Comparing FlatMapGroupsWithState (FMGWS) and TransformWithState (TWS) (1M rows/batch).

5.2.1 Performance evaluation. We compared the throughput of transformWithState (TWS) and flatMapGroupsWithState (FMGWS) for state types such as ValueState, ListState, MapState and languages such as Scala and Python across single-value updates, topK use-cases, list appends, map lookups, and other workloads.

Benchmark setup. For the benchmarks, we generated test data with different throughput rates (10k rows/batch to 1M rows/batch) and key cardinalities. All benchmarks were run on 5 node clusters (1 driver node and 5 executor nodes) using r5d.2xl instances with 8 cores and 64GB memory each. All runs were executed for at least 30 minutes, and multiple runs were included for each test; we report the mean findings below. All benchmarks used RocksDB as the state store provider and 40 shuffle partitions.

Results. For the single value based update benchmark (Figure 9a), TWS performs comparably to FMGWS (up to 1.05x faster in Scala and 1.03x in Python).

For the topK use case (Figure 9b), TWS performance is also generally comparable to FMGWS across K values.

For the list-append-based use case (Figure 9c), TWS shows superior performance, particularly with medium to large list sizes (e.g., 1K elements). Specifically, performance improved by up to 3.42x in Scala and 2x in Python, with gains improving linearly with the cardinality of list elements.

For the map-lookup-based use case (Figure 9d), TWS is significantly more performant when the number of unique user keys (N) per grouping key is high, as TWS only performs constant state store operations compared to FMGWS's single operation which requires costly full map serialization/deserialization (e.g., Scala TWS up to 1.5x FMGWS throughput). The gains improve linearly with the cardinality of user keys. We also found that FMGWS is slightly more performant when N is small, as the overhead of multiple TWS state store operations outweighs the FMGWS serialization/deserialization cost (e.g., Scala TWS down to 0.69x FMGWS throughput).

Takeaways. We found that the new transformWithState APIs significantly outperformed the existing APIs for moderate to large list sizes for append use-cases (up to 3.4x faster) since we don't need to pay the cost of performing serialization/deserialization on each update. Similarly, for map-based lookups, we found that the new APIs are more performant (up to 1.5x for tested cardinality) compared to existing APIs for medium to high cardinality of user keys since point lookups and updates only perform constant state store operations instead of serializing and writing the whole map each time. We saw that for small list and map sizes, the performance is similar or slightly worse compared to the existing APIs since the state store updates tend to outweigh the serialization costs. For simple value-based updates, we saw that performance was largely in line with or slightly better than existing APIs, as expected. Additionally, we saw that just within a year from launch of this new API, only 53% of unique customers relied exclusively on the legacy API, 37% chose to use only the new API and 10% chose to run both, suggesting strong traction and adoption of the new APIs.

6 SECURITY AND GOVERNANCE

6.1 The streaming governance gap

While Databricks Lakeguard [24] successfully secures ephemeral batch workloads (see § 2.6), the standard request-response model encounters unique challenges in the context of Structured Streaming that remain unexplored in the prior literature. Unlike batch jobs, streaming requires active interaction during execution to poll for status and issue commands, as well as handling Streaming

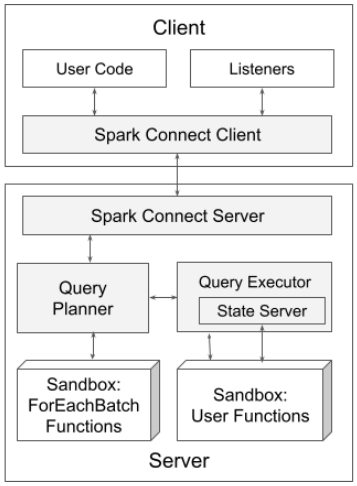


Figure 10: Streaming workflow with Lakeguard.

QueryListener and foreachBatch. Furthermore, executing arbitrary stateful operators requires frequent data exchange between the engine and user logic.

6.2 Streaming query lifecycle

To execute a streaming query, the user invokes .start(), prompting the local Spark Connect client to normalize and forward the plan to the server. The server provisions a (QueryID, RunId) pair, which the client persists in a query handle used for issuing management RPCs (e.g., status checks or termination).

Similar to batch workloads, the lifecycle of a streaming query is bound to the Spark Connect session. If a session expires, all associated streams are automatically terminated and cleaned up. Within an active session, the server maintains the necessary context to facilitate client interactions by maintaining a mapping from (QueryID, RunId) to active query instances. Session timeouts trigger resource reclamation. The server retains the streaming query with a retention policy, to allow the client to inspect the final status, exceptions, or progress metrics.

6.3 Challenges for user-defined logic

A central challenge in Spark Connect is determining the execution site for arbitrary user logic. We evaluated two primary models: Server-Side Execution, where functions run in isolated containers (similar to UDFs), and Client-Side Execution, where the client acts as a remote worker via RPCs. We adopted distinct strategies for StreamingQueryListener and the foreachBatch (§ 2.5) operator based on their synchronization requirements.

StreamingQueryListener employs Client-Side Execution. This matches its role as an asynchronous observability callback: transient delivery delays affect monitoring freshness but not pipeline progress, and the client is typically where users or platforms consume and report status and metrics. Running listener logic inside the Spark server would add unnecessary barriers to observability integration. The server therefore pushes events (e.g., onQueryProgress) to the client via a long-lived gRPC channel (*Server* →

Client), except onQueryStarted, which is a special case and is sent with the start() response to preserve API ordering guarantees.

Conversely, foreachBatch adopts Server-Side Execution. Since its logic is synchronous and blocking—a query cannot advance until the batch returns—running it on a separate client node would make query progress depend on remote communication and client-node availability, introducing avoidable latency and failure modes. Instead, the client serializes the function (via CloudPickle) to the server, which provisions an isolated Python worker to execute the logic securely co-located with the engine.

This server-side execution model uses a nested architecture. Because the isolated Python worker cannot access the JVM directly—yet needs to submit queries back to the Spark Engine, potentially multiple times per batch—it operates as a secondary, standalone Spark Connect client. The JVM-side Spark Connect Server opens a dedicated internal RPC endpoint. To guarantee consistency, the system employs Dual-Layered Session Isolation where the core streaming engine clones the root SparkSession to enforce configurations specific to this query and isolate session-specific artifacts (such as temporary views defined within foreachBatch itself), while the Spark Connect Server provisions a distinct session for the worker to multiplex RPC commands without colliding with the main client's state.

6.4 Stateful operators and state store management

Arbitrary stateful operators, such as flatMapGroupsWithState and transformWithState (§ 5.2), introduce complex challenges in a sandboxed environment. Unlike simple UDFs that process stateless inputs and outputs, these operators require frequent read and write access to the state store.

Latency and architecture alignment. Separating the Spark engine from the execution sandbox inevitably introduces latency. Fortunately, Spark Structured Streaming's micro-batch architecture aligns well with this constraint; by batching inputs and outputs, the cross-boundary communication overhead can be amortized effectively.

Security concerns. Arbitrary stateful operators execute user code, introducing distinct security challenges in multi-tenant environments. Query state is maintained in two locations: on local disks for immediate read and write operations, and in object storage for durable checkpoints. To secure these checkpoints, we leverage cloud-native encryption at rest and in transit (such as in S3[5]), and strictly enforce access control via Unity Catalog ACLs [15].

Furthermore, to prevent both cross-tenant data leaks and engine instability, user code runs under sandbox isolation [24]; it cannot directly access state store data in memory or on disk. All state store access is mediated by the Spark engine.

State store placement. A critical design decision concerns the location of the state store. Placing the state store within the sandbox is suboptimal for two reasons:

(1) **Resource constraints:** Containers are designed to be lightweight and ephemeral.
(2) **Security:** As mentioned above, untrusted user code shouldn't have direct access to state stores.

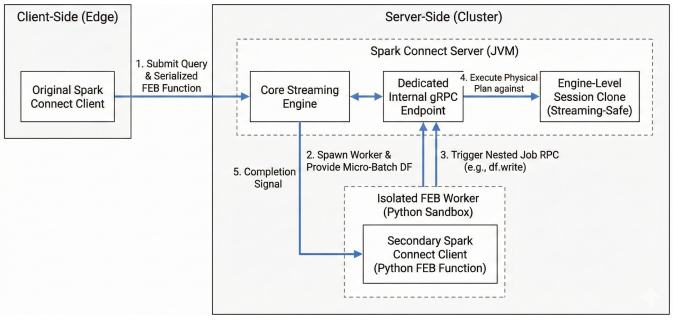


Figure 11: Spark Connect forEachBatch (FEB) execution.

We therefore keep the state store inside the Spark engine, so sandboxed code communicates with the engine for all state operations.

IPC Amortization and the Throughput-Latency Trade-off. transformWithState enables complex state management through fine grained primitives (ValueState, ListState, MapState) and timers. Unlike the monolithic state model, it employs a lazy, fine-grained interaction pattern where state is fetched from the engine only when explicitly accessed. While this supports massive state sizes, it generates frequent, small I/O operations across the process boundary.

The micro-batch architecture automatically facilitates IPC amortization, provided that a single batch contains multiple records for the same key. For instance, consider a query analyzing user activity where a user generates an event every second. A longer batch interval will capture more records for that specific user, allowing the system to reuse fetched state across multiple events. This drastically reduces the number of IPC round-trips required per record, lowering the average cost. This dynamic creates a fundamental throughput-latency trade-off: maximizing throughput requires accumulating larger batches to ensure multiple records per key are grouped together, which inevitably increases end-to-end latency.

To further mitigate overhead, the system employs write batching, which buffers state updates (such as put or remove) and flushes them to the engine in a single bulk request. Additionally, parallel execution is used to process independent callbacks or data partitions simultaneously, effectively hiding communication latency behind concurrent processing. Despite these optimizations, minimizing latency for complex stateful logic in isolated environments remains an open research area.

Overall, across both user-defined logic and transformWithState, our measurements show that the sandbox performance overhead for streaming queries is on par with the batch workloads previously detailed in the Lakeguard paper [24]. The Spark Connect protocol– which standardizes both the client and server communication, and between the sandbox and the protected Spark engine—is fully open-sourced in Apache Spark, while Databricks provides its own secure sandbox implementation.

7 RELATED WORK

Stream processing systems. Stream processing systems commonly use two primary execution models: continuous operator

and micro-batch. Systems like Apache Flink [14] and Google Cloud Dataflow [4] employ the continuous model, where long-lived operators process records individually. While this yields low latency, it requires complex distributed snapshotting (e.g., Chandy-Lamport [16]) for consistency. This model is further operationalized by managed services like Ververica and Confluent, the latter commercializing the library-centric approach of Apache Kafka Streams [39] for microservices.

Conversely, Spark Streaming [41] and Structured Streaming [11] pioneered the micro-batch architecture. While historically criticized for higher latency [27], this paradigm has evolved significantly. Research such as *Drizzle* [38] introduced group scheduling to mitigate overhead, while other works have optimized throughput via adaptive batch sizing [18].

Pipelining. Hiding latency through overlapping execution phases is a concept deeply rooted in microprocessor design [26]. Within data management, it has successfully transitioned into software via pipelined query execution [23, 29] and continuous stream processing engines like Aurora [1]. Our work extends this lineage by shifting the application of pipelining from operator-level dataflows to the micro-batch lifecycle itself. By modeling the "plan-execute-commit" cycle as a staged software pipeline, we resolve the resource underutilization inherent in synchronous boundaries without sacrificing the fault-tolerance advantages of the BSP execution model.

State management and observability. Effective state management is critical for complex event processing. Apache Samza pioneered the two-tier architecture, adopted the changelog, and used RocksDB [30]. Flink also adopted this approach [8, 14]. However, observing distributed state remains challenging. Flink's State Processor API allows offline access to savepoints, serving a similar purpose to our *State Data Source Reader*, which exposes the "black box" of streaming state as a queryable entity. Furthermore, our transformWithState operator advances the expressiveness of stateful logic in Spark, offering composite types and timer management that align with the capabilities found in Flink's ProcessFunction, but integrated natively into Spark's declarative DataFrame API.

Ingestion. Achieving exactly-once semantics typically requires replayable sources and idempotent sinks [35]. Traditional connectors rely on offset tracking to define batch boundaries. However, systems like Google Cloud Pub/Sub and AWS Kinesis often lack strict offset semantics. Previous approaches in MillWheel [3] handled this via rigorous watermark tracking and deduplication. Our work extends the connector ecosystem to handle these sources by implementing *asynchronous pre-fetching* and *state-based deduplication* directly within the driver-executor model, ensuring exactly-once processing even without native upstream offsets.

Security. Secure multi-tenancy in data processing traditionally relies on containerization (e.g., Kubernetes) or microVMs (e.g., Firecracker [2]) to isolate user logic. However, the overhead of cold starts makes strict isolation difficult for low latency streaming. While batch systems often use a Client/Server architecture to separate the control plane from user code (e.g., PySpark's use of separate Python processes), this often introduces serialization bottlenecks. Recent work investigates WebAssembly (Wasm) [25] as a lightweight sandboxing mechanism. Our *Lakeguard* architecture adopts a hybrid approach: it isolates untrusted operations (UDFs) in restricted sandboxes while maintaining high-throughput access to the execution engine. Although designed for different applications, Cloudburst [33] tackles a similar challenge of mitigating state-access latency for isolated, untrusted code. Within the Apache Spark ecosystem, securing production data pipelines traditionally requires physical, cluster-level isolation for strict multi-tenancy, as is common practice in AWS EMR [6]. In-engine authorization tools like Apache Ranger struggle with imperative User-Defined Functions (UDFs) executing directly on the JVM, often forcing restrictive SQL-only interfaces to maintain security [34]. Apache Kyuubi circumvents this by provisioning isolated user engines as a multi-tenant gateway [9]. Beyond Spark, Apache Flink deployments secure multi-tenancy via costly physical clusters, and isolating their UDFs requires high-overhead external gRPC runtimes [17]. For a more extensive discussion of these related works, we refer readers to the Lakeguard paper [24].

8 CONCLUSION

Over the past decade, Apache Spark Structured Streaming has transitioned from an early streaming platform to become the critical backbone of enterprise data processing. While the industry often debated the trade-offs between continuous processing and micro-batching, our experience has proven that the microbatch model can provide a solid foundation for the efficiency, scale, fault tolerance, and exactly-once semantics that production environments demand. In this paper, we highlighted four areas in which we improved the system- efficiency through microbatch pipelining, diverse connectors, new stateful operators like transformWithState, and zero-trust security. Structured Streaming stands as a testament to the power of a resilient architecture evolved through production scale, ready to define the next decade of real-time data and AI workloads.

ACKNOWLEDGMENTS

We thank the diverse Apache Spark developer community that has contributed to Structured Streaming, Spark Streaming and Spark SQL over the years. We thank the following streaming team members for their contributions: Pranav Anand, Raghu Angadi, Alex Balikov, Zerui Bao, Tathagata Das, Dymtro Fedoriaka, Zifei Feng, Praveen Gattu, Zikang Han, Arsenii Krasikov, Chaoqin Li, Wei Liu, Gurpreet Nanda, Neil Ramaswamy, Bhuwan Sahni, Brook Walls, Dylan Wong, Min Yang, David Young, Burak Yavuz, Jing Zhan, Peng Zhong, You Zhou, and Ryan Zhu. We thank Daniel Tenedorio for reviewing the paper and providing feedback.

REFERENCES

[1] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *The VLDB Journal* 12, 2 (2003), 120–139.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, et al. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 419–434.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, et al. 2013. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (2013), 1033–1044.

[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, et al. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale Unbounded, Out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (2015), 1792–1803.

[5] Amazon AWS. [n.d.]. Protecting data with encryption. https://docs.aws.amazon.com/AmazonS3/latest/userguide/UsingEncryption.html Accessed: 2026-07-06.

[6] Amazon Web Services. 2022. Design considerations for Amazon EMR on EKS in a multi-tenant Amazon EKS environment. https://aws.amazon.com/blogs/big-data/design-considerations-for-amazon-emr-on-eks-in-a-multi-tenant-amazon-eks-environment/ Accessed: 2026-07-07.

[7] Amazon Web Services. 2026. Amazon Kinesis Data Streams Terminology and concepts. https://docs.aws.amazon.com/streams/latest/dev/key-concepts.html#enabled-application Accessed: 2026-02-27.

[8] Apache Flink Community. 2021. FLIP-158: Generalized incremental checkpoints. https://cwiki.apache.org/confluence/display/FLINK/FLIP-158%3A+Generalized+incremental+checkpoints. Accessed: 2026-02-27.

[9] Apache Software Foundation. 2023. *Welcome to Apache Kyuubi: A Unified Multi-tenant JDBC Interface.* https://kyuubi.readthedocs.io/ Accessed: 2026-07-07.

[10] Michael Armbrust, Tathagata Das, Liwen Sun, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Peter Boncz, Mostafa Mokhtar, Herman van Hövell, 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. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424. https://www.vldb.org/pvldb/vol13/p3411-armbrust.pdf Accessed: 2026-07-07.

[11] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 601–613. https://doi.org/10.1145/3183713.3190664 Accessed: 2026-07-07.

[12] 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*. ACM, 1383–1394.

[13] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Harish Kommareddy, Krishnan Maheshwari, Peter Mattis, Victor Smirnov, Kaushik Ravichandran, Michael Ryan, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2326–2339. https://doi.org/10.1145/3514221.3526054 Accessed: 2026-07-07.

[14] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 38, 4 (2015).

[15] Ramesh Chandra et al. 2025. Unity Catalog: Open and Universal Governance for the Lakehouse and Beyond. In *Companion of the 2025 International Conference on Management of Data (SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 310–322. https://doi.org/10.1145/3722212.3724459 Accessed: 2026-07-07.

[16] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (1985), 63–75.

[17] Conduktor. 2024. Multi-Tenancy in Kafka and Flink Environments: Logical vs Physical Separation. https://www.conduktor.io/glossary/multi-tenancy-in-kafka-environments Accessed: 2026-07-07.

[18] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. 2014. Adaptive Stream Processing using Dynamic Batch Sizing. In *Proceedings of the 2014 ACM Symposium on Cloud Computing (SoCC)*. ACM, 1–13.

[19] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI)*. 10.

[20] Delta Lake Project. [n.d.]. Delta Deletion Vectors. https://docs.delta.io/delta-deletion-vectors/. Accessed: 2026-03-01.

[21] Google Cloud. [n.d.]. Cloud Pub/Sub: A fully-managed real-time messaging service. https://cloud.google.com/pubsub Accessed: 2026-03-02.

[22] Google Cloud. 2026. Subscription overview – Default subscription properties. https://docs.cloud.google.com/pubsub/docs/subscription-overview#default_properties Accessed: 2026-02-27.

[23] Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)* 25, 2 (1993), 73–169.

[24] 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* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 418–430. https://doi.org/10.1145/3722212.3724433 Accessed: 2026-07-07.

[25] B. L. Hall, N. Ramachandran, A. Kishore, et al. 2020. A Survey of WebAssembly for Serverless Computing. *arXiv preprint arXiv:2010.07115* (2020).

[26] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

[27] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, et al. 2018. Benchmarking Distributed Stream Data Processing Systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1507–1518.

[28] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB)*. Athens, Greece, 1–7.

[29] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

[30] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. 2017. Samza: stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1634–1645.

[31] Anton Okolnychyi, Cheng Sun, Kazuyuki Tanimura, et al. 2024. Petabyte-Scale Row-Level Operations in Data Lakehouses. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4159–4172. https://vldb.org/pvldb/vol17/p4159-okolnychyi.pdf Accessed: 2026-07-07.

[32] Anish Shrigondekar. 2023. SPIP: Structured Streaming - Arbitrary State API v2. Apache Spark Developer Mailing List. https://lists.apache.org/thread/3jyjdgk1m5zyqfmrocnt6t415703nc8l Proposing transformWithState for Spark Structured Streaming. Available at https://lists.apache.org/thread/3jyjdgk1m5zyqfmrocnt6t415703nc8l. Accessed: July 6, 2026.

[33] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proceedings of the VLDB Endowment* 13, 11 (2020), 2438–2452.

[34] Starburst Data. 2022. *Apache Ranger Overview and Limitations.* https://docs.starburst.io/latest/security/ranger-overview.html Accessed: 2026-07-07.

[35] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 Requirements of Real-Time Stream Processing. *SIGMOD Rec.* 34, 4 (2005), 42–47.

[36] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, et al. 2014. Storm@Twitter. In *Proceedings of the 2014 International Conference on Management of Data (SIGMOD)*. ACM, 147–156.

[37] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111.

[38] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, et al. 2017. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. ACM, 374–389.

[39] Guozhang Wang, Joel Koshy, Sriram Subramanian, et al. 2015. Building a Replicated Logging System with Apache Kafka. *Proc. VLDB Endow.* 8, 12 (2015), 1654–1655.

[40] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, 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 conference on Networked Systems Design and Implementation*. USENIX Association, 15–28.

[41] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 423–438.