Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark

Michael Armbrust[†], Tathagata Das[†], Joseph Torres[†], Burak Yavuz[†], Shixiong Zhu[†], Reynold Xin[†], Ali Ghodsi[†], Ion Stoica[†], Matei Zaharia[†‡]
[†]Databricks Inc., [‡]Stanford University

Abstract

With the ubiquity of real-time data, organizations need streaming systems that are scalable, easy to use, and easy to integrate into business applications. Structured Streaming is a new high-level streaming API in Apache Spark based on our experience with Spark Streaming. Structured Streaming differs from other recent streaming APIs, such as Google Dataflow, in two main ways. First, it is a purely *declarative* API based on automatically incrementalizing a static relational query (expressed using SQL or DataFrames), in contrast to APIs that ask the user to build a DAG of physical operators. Second, Structured Streaming aims to support *end-to-end* real-time applications that integrate streaming with batch and interactive analysis. We found that this integration was often a key challenge in practice. Structured Streaming achieves high performance via Spark SQL's code generation engine and can outperform Apache Flink by up to 2× and Apache Kafka Streams by 90×. It also offers rich operational features such as rollbacks, code updates, and mixed streaming/batch execution. We describe the system's design and use cases from several hundred production deployments on Databricks, the largest of which process over 1 PB of data per month.

ACM Reference Format:
M. Armbrust et al.. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3183713.3190664

1 Introduction

Many high-volume data sources operate in real time, including sensors, logs from mobile applications, and the Internet of Things. As organizations have gotten better at capturing this data, they also want to process it in real time, whether to give human analysts the freshest possible data or drive automated decisions. Enabling broad access to streaming computation requires systems that are scalable, easy to use and easy to integrate into business applications.

While there has been tremendous progress in distributed stream processing systems in the past few years [2, 15, 17, 27, 32], these systems still remain fairly challenging to use in practice. In this paper, we begin by describing these challenges, based on our experience with Spark Streaming [37], one of the earliest stream processing systems to provide a high-level, functional API. We found that two challenges frequently came up with users. First, streaming systems often ask users to think in terms of complex physical execution concepts, such as at-least-once delivery, state storage and triggering modes, that are unique to streaming. Second, many systems focus *only* on streaming computation, but in real use cases, streaming is often part of a larger business application that also includes batch analytics, joins with static data, and interactive queries. Integrating streaming systems with these other workloads (e.g., maintaining transactionality) requires significant engineering effort.

Motivated by these challenges, we describe Structured Streaming, a new high-level API for stream processing that was developed in Apache Spark starting in 2016. Structured Streaming builds on many ideas in recent stream processing systems, such as separating processing time from event time and triggers in Google Dataflow [2], using a relational execution engine for performance [12], and offering a language-integrated API [17, 37], but aims to make them simpler to use and integrated with the rest of Apache Spark. Specifically, Structured Streaming differs from other widely used open source streaming APIs in two ways:

- **Incremental query model:** Structured Streaming automatically incrementalizes queries on static datasets expressed through Spark's SQL and DataFrame APIs [8], meaning that users typically only need to understand Spark's batch APIs to write a streaming query. Event time concepts are especially easy to express and understand in this model. Although incremental query execution and view maintenance are well studied [11, 24, 29, 38], we believe Structured Streaming is the first effort to adopt them in a widely used open source system. We found that this incremental API generally worked well for both novice and advanced users. For example, advanced users can use a set of stateful processing operators that give fine-grained control to implement custom logic while fitting into the incremental model.
- **Support for end-to-end applications:** Structured Streaming's API and built-in connectors make it easy to write code that is "correct by default" when interacting with external systems and can be integrated into larger applications using Spark and other software. Data sources and sinks follow a simple transactional model that enables "exactly-once" computation by default. The incrementalization based API naturally makes it easy to run a streaming query as a batch job or develop hybrid applications that join streams with static data computed through Spark's batch APIs. In addition, users can manage multiple streaming queries dynamically and run interactive queries on consistent

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3190664

snapshots of stream output, making it possible to write applications that go beyond computing a fixed result to let users refine and drill into streaming data.

Beyond these design decisions, we made several other design choices in Structured Streaming that simplify operation and increase performance. First, Structured Streaming reuses the Spark SQL execution engine [8], including its optimizer and runtime code generator. This leads to high throughput compared to other streaming systems (e.g., 2× the throughput of Apache Flink and 90× that of Apache Kafka Streams in the Yahoo! Streaming Benchmark [14]), as in Trill [12], and also lets Structured Streaming automatically leverage new SQL functionality added to Spark. The engine runs in a microbatch execution mode by default [37] but it can also use a low-latency continuous operators for some queries because the API is agnostic to execution strategy [6].

Second, we found that operating a streaming application can be challenging, so we designed the engine to support failures, code updates and recomputation of already outputted data. For example, one common issue is that new data in a stream causes an application to crash, or worse, to output an incorrect result that users do not notice until much later (e.g., due to mis-parsing an input field). In Structured Streaming, each application maintains a write-ahead event log in human-readable JSON format that administrators can use to restart it from an arbitrary point. If the application crashes due to an error in a user-defined function, administrators can update the UDF and restart from where it left off, which happens automatically when the restarted application reads the log. If the application was outputting incorrect data instead, administrators can manually roll it back to a point before the problem started and recompute its results starting from there.

Our team has been running Structured Streaming applications for customers of Databricks' cloud service since 2016, as well as using the system internally, so we end the paper with some example use cases. Production applications range from interactive network security analysis and automated alerts to incremental Extract, Transform and Load (ETL). Users often leverage the design of the engine in interesting ways, e.g., by running a streaming query "discontinuously" as a series of single-microbatch jobs to leverage Structured Streaming's transactional input and output without having to pay for cloud servers running 24/7. The largest customer applications we discuss process over 1 PB of data per month on hundreds of machines. We also show that Structured Streaming outperforms Apache Flink and Kafka Streams by 2× and 90× respectively in the widely used Yahoo! Streaming Benchmark [14].

The rest of this paper is organized as follows. We start by discussing the stream processing challenges reported by users in Section 2. Next, we give an overview of Structured Streaming (Section 3), then describe its API (Section 4), query planning (Section 5), execution (Section 6) and operational features (Section 7). In Section 8, we describe several large use cases at Databricks and its customers. We then measure the system's performance in Section 9, discuss related work in Section 10 and conclude in Section 11.

2 Stream Processing Challenges

Despite extensive progress in the past few years, distributed streaming applications are still generally considered difficult to develop and operate. Before designing Structured Streaming, we spent time discussing these challenges with users and designers of other streaming systems, including Spark Streaming, Truviso, Storm, Dataflow and Flink. This section details the challenges we saw.

2.1 Complex and Low-Level APIs

Streaming systems were invariably considered more difficult to use than batch ones due to complex API semantics. Some complexity is to be expected due to new concerns that arise only in streaming: for example, the user needs to think about what type of intermediate results the system should output before it has received all the data relevant to a particular entity, e.g., to a customer's browsing session on a website. However, other complexity arises due to the *low-level* nature of many streaming APIs: these APIs often ask users to specify applications at the level of *physical operators* with complex semantics instead of a more declarative level.

As a concrete example, the Google Dataflow model [2] has a powerful API with a rich set of options for handling event time aggregation, windowing and out-of-order data. However, in this model, users need to specify a windowing mode, triggering mode and trigger refinement mode (essentially, whether the operator outputs deltas or accumulated results) for each aggregation operator. Adding an operator that expects deltas after an aggregation that outputs accumulated results will lead to unexpected results. In essence, the raw API [10] asks the user to write a physical operator graph, not a logical query, so every user of the system needs to understand the intricacies of incremental processing.

Other APIs, such as Spark Streaming [37] and Flink's DataStream API [18], are also based on writing DAGs of physical operators and offer a complex array of options for managing state [20]. In addition, reasoning about applications becomes even more complex in systems that relax exactly-once semantics [32], effectively requiring the user to design and implement a consistency model.

To address this issue, we designed Structured Streaming to make simple applications simple to express using its incremental query model. In addition, we found that adding customizable *stateful processing* operators to this model still enabled advanced users to build their own processing logic, such as custom session-based windows, while staying within the incremental model (e.g., these same operators also work in batch jobs). Other open source systems have also recently added incremental SQL queries [15, 19], and of course databases have long supported them [11, 24, 29, 38].

2.2 Integration in End-to-End Applications

The second challenge we found was that nearly every streaming workload must run in the context of a larger application, and this integration often requires significant engineering effort. Many streaming APIs focus primarily on reading streaming input from a source and writing streaming output to a sink, but end-to-end business applications need to perform other tasks. Examples include:

(1) The business purpose of the application may be to enable interactive queries on fresh data. In this case, a streaming job is used to update summary tables in a structured storage system such as an RDBMS or Apache Hive [33]. It is important that when the streaming job updates its result, it does so atomically, so users do not see partial results. This can be difficult with file-based big data systems like Hive, where tables are partitioned across files, or even with parallel loads into a data warehouse.

(2) An Extract, Transform and Load (ETL) job might need to join a stream with static data loaded from another storage system or transformed using a batch computation. In this case, it is important to be able to reason about consistency across the two systems (e.g., what happens when the static data is updated?), and it is useful to write the whole computation in a single API.

(3) A team may occasionally need to run its streaming business logic as a batch application, e.g., to backfill a result on old data or test alternate versions of the code. Rewriting the code in a separate system would be time-consuming and error-prone.

We address this challenge by integrating Structured Streaming closely with Spark's batch and interactive APIs.

2.3 Operational Challenges

One of the largest challenges to deploying streaming applications in practice is management and operation. Some key issues include:

- **Failures:** This is the most heavily studied issue in the research literature. In addition to single node failures, systems also need to support graceful shutdown and restart of the whole application, e.g., to let operators migrate it to a new cluster.
- **Code Updates:** Applications are rarely perfect, so developers may need to update their code. After an update, they may want the application to restart where it left off, or possibly to *recompute* past results that were erroneous due to a bug. Both cases need to be supported in the streaming system's state management and fault recovery mechanisms. Systems should also support updating the runtime itself (e.g., patching Spark).
- **Rescaling:** Applications see varying load over time, and generally *increasing* load in the long term, so operators may want to scale them up and down dynamically, especially in the cloud. Systems based on a static communication topology, while conceptually simple, are difficult to scale dynamically.
- **Stragglers:** Instead of outright failing, nodes in the streaming system can slow down due to hardware or software issues and degrade the throughput of the whole application. Systems should automatically handle this situation.
- **Monitoring:** Streaming systems need to give operators clear visibility into system load, backlogs, state size and other metrics.

2.4 Cost and Performance Challenges

Beyond operational and engineering issues, the cost-performance of streaming applications can be an obstacle because these applications run 24/7. For example, without dynamic rescaling, an application will waste resources outside peak hours; and even with rescaling, it may be more expensive to compute a result continuously than to run a periodic batch job. We thus designed Structured Streaming to leverage all the execution optimizations in Spark SQL [8].

So far, we chose to optimize *throughput* as our main performance metric because we found that it was often the most important metric in large-scale streaming applications. Applications that require a *distributed* streaming system usually work with large data volumes coming from external sources (e.g., mobile devices, sensors or IoT), where data may already incur a delay just getting to the system. This is one reason why event time processing is an important feature in these systems [2]. In contrast, latency-sensitive applications

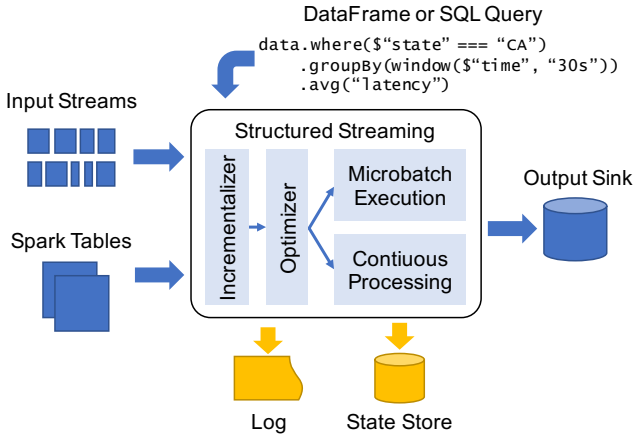


Figure 1: The components of Structured Streaming.

such as high-frequency trading or physical system control loops often run on a single scale-up processor, or even custom hardware like ASICs and FPGAs [3]. However, we also designed Structured Streaming to support executing over latency-optimized engines, and implemented a continuous processing mode for this task, which we describe in Section 6.3. This is a change over Spark Streaming, where microbatching was "baked into" the API.

3 Structured Streaming Overview

Structured Streaming aims to tackle the stream processing challenges we identified through a combination of API and execution engine design. In this section, we give a brief overview of the overall system. Figure 1 shows Structured Streaming's main components.

Input and Output. Structured Streaming connects to a variety of *input sources* and *output sinks* for I/O. To provide "exactly-once" output and fault tolerance, it places two restrictions on sources and sinks, which are similar to other exactly-once systems [17, 37]:

(1) Input sources must be *replayable*, allowing the system to re-read recent input data if a node crashes. In practice, organizations use a reliable message bus such as Amazon Kinesis or Apache Kafka [5, 23] for this purpose, or simply a durable file system.

(2) Output sinks must support *idempotent* writes, to ensure reliable recovery if a node fails while writing. Structured Streaming can also provide *atomic* output for certain sinks that support it, where the entire update to the job's output appears atomically even if it was written by multiple nodes working in parallel.

In addition to external systems, Structured Streaming also supports input and output from tables in Spark SQL. For example, users can compute a static table from any of Spark's batch input sources and join it with a stream, or ask Structured Streaming to output to an in-memory Spark table that users can query interactively.

API. Users program Structured Streaming by writing a query against one or more streams and tables using Spark SQL's batch APIs: SQL and DataFrames [8]. This query defines an *output table* that the user wants to compute, assuming that each input stream is replaced by a table holding all the data received from that stream so far. The engine then determines how to compute and write this

output table into a sink *incrementally*, using similar techniques to incremental view maintenance [11, 29]. Different sinks also support different *output modes*, which determine how the system may write out its results: for example, some sinks are append-only by nature, while others allow updating records in place by key.

To support streaming specifically, Structured Streaming also adds several API features that fit in the existing Spark SQL API:

(1) *Triggers* control how often the engine will attempt to compute a new result and update the output sink, as in Dataflow [2].

(2) Users can mark a column as denoting *event time* (a timestamp set at the data source), and set a *watermark* policy to determine when enough data has been received to output a result for a specific event time, as in [2].

(3) *Stateful operators* allow users to track and update mutable state by key in order to implement complex processing, such as custom session-based windows. These are similar to Spark Streaming's `updateStateByKey` API [37].

Note that windowing, another key feature for streaming, is done using Spark SQL's existing aggregation APIs. In addition, all the new APIs added by Structured Streaming also work in batch jobs.

Execution. Once it has received a query, Structured Streaming optimizes it, incrementalizes it, and begins executing it. By default, the system uses a microbatch model similar to Discretized Streams in Spark Streaming, which supports dynamic load balancing, rescaling, fault recovery and straggler mitigation by dividing work into small tasks [37]. In addition, it can use a continuous processing mode based on traditional long-running operators (Section 6.3).

In both cases, Structured Streaming uses two forms of durable storage to achieve fault tolerance. First, a *write-ahead log* keeps track of which data has been processed and reliably written to the output sink from each input source. For some output sinks, this log can be integrated with the sink to make updates to the sink atomic. Second, the system uses a larger-scale *state store* to hold snapshots of operator states for long-running aggregation operators. These are written asynchronously, and may be "behind" the latest data written to the output sink. The system will automatically track which state it has last updated in its log, and recompute state starting from that point in the data on failure. Both the log and state store can run over pluggable storage systems (e.g., HDFS or S3).

Operational Features. Using the durability of the write-ahead log and state store, users can achieve several forms of rollback and recovery. An *entire* Structured Streaming application can be shut down and restarted on new hardware. Running applications also tolerate node crashes, additions and stragglers automatically, by sending tasks to new nodes. For code updates to UDFs, it is sufficient to stop and restart the application, and it will begin using the new code. In addition, users can manually roll back the application to a previous point in the log and redo the part of the computation starting then, beginning from an older snapshot of the state store.

In addition, Structured Streaming's ability to execute with microbatches lets it "adaptively batch" data so that it can quickly catch up with input data if the load spikes or if a job is rolled back, then return to low latency later. This makes operation significantly simpler (e.g., users can safely update job code more often).

The next sections go into detail about Structured Streaming's API (§4), query planning (§5) and job execution and operation (§6).

4 Programming Model

Structured Streaming combines elements of Google Dataflow [2], incremental queries [11, 29, 38] and Spark Streaming [37] to enable stream processing beneath the Spark SQL API. In this section, we start by showing a short example, then describe the semantics of the model and the streaming-specific operators we added in Spark SQL to support streaming use cases (e.g., stateful operators).

4.1 A Short Example

Structured Streaming operates within Spark's structured data APIs: SQL, DataFrames and Datasets [8]. The main abstraction users work with is *tables* (represented by the DataFrames or Dataset classes), which each represent a view to be computed from input sources to the system.[1] When users create a table/DataFrame from a streaming input source, and attempt to compute it, Spark will automatically launch a streaming computation.

As a simple example, let us start with a *batch* job that counts clicks by country of origin for a web application. Suppose that the input data is JSON files and the output should be Parquet. This job can be written with Spark DataFrames in Scala as follows:

```
// Define a DataFrame to read from static data
data = spark.read.format("json").load("/in")

// Transform it to compute a result
counts = data.groupBy($"country").count()

// Write to a static data sink
counts.write.format("parquet").save("/counts")
```

Changing this job to use Structured Streaming only requires modifying the input and output sources, not the transformation in the middle. For example, if new JSON files are going to *continually* be uploaded to the `/in` directory, we can modify our job to continually update `/counts` by changing only the first and last lines:

```
// Define a DataFrame to read streaming data
data = spark.readStream.format("json").load("/in")

// Transform it to compute a result
counts = data.groupBy($"country").count()

// Write to a streaming data sink
counts.writeStream.format("parquet")
      .outputMode("complete").start("/counts")
```

The output mode parameter on the sink here specifies how Structured Streaming should update the sink. In this case, the complete mode means to write a complete result file for each update, because the file output sink chosen does not support fine-grained updates. However, other sinks, such as key-value stores, support additional output modes (e.g., updating just the changed keys).

Under the hood, Structured Streaming will automatically incrementalize the query specified by the transformation(s) from input sources to data sinks, and execute it in a streaming fashion. The

[1] Spark SQL offers several slightly different APIs that map to the same query engine. The DataFrame API, modeled after data frames in R and Pandas [28, 30], offers a simple interface to build relational queries programmatically that is familiar to many users. The Dataset API adds static typing over DataFrames, similar to RDDs [36]. Alternatively, users can write SQL directly. All APIs produce a relational query plan.

engine will also automatically maintain state and checkpoint it to external storage as needed—in this case, for example, we have a running count aggregation since the start of the stream, so the engine will keep track of the running counts for each country.

Finally, the API also naturally supports windowing and event time through Spark SQL's existing aggregation operators. For example, instead of counting data by country, we could count it in 1-hour sliding windows advancing every 5 minutes by changing just the middle line of the computation as follows:

```
// Count events by windows on the "time" field
data.groupBy(window($"time","1h","5min")).count()
```

The `time` field here (event time) is just a field in the data, similar to `country` earlier. Users can also set a watermark on this field to let the system forget state for old windows after a timeout (§4.3.1).

4.2 Programming Model Semantics

Formally, we define the semantics of Structured Streaming's programming model as follows:

(1) Each input source provides a partially ordered set of records over time. We assume partial orders here because some message bus systems are parallel and do not define a total order across records—for example, Kafka divides streams into "partitions" that are each ordered.

(2) The user provides a query to execute across the input data that can output a *result table* at any given point in processing time. Structured Streaming will always produce results consistent with running this query on a *prefix of the data in all input sources*. That is, it will never show results that incorporate one input record but do not incorporate its ancestors in the partial order. Moreover, these prefixes will be increasing over time.

(3) *Triggers* tell the system *when* to run a new incremental computation and update the result table. For example, in microbatch mode, the user may wish to trigger an incremental update every minute (in processing time).

(4) The sink's *output mode* specifies how the result table is written to the output system. The engine supports three distinct modes:

- *Complete:* The engine writes the whole result table at once, e.g., replacing a whole file in HDFS with a new version. This is of course inefficient when the result is large.
- *Append:* The engine can only add records to the sink. For example, a map-only job on a set of input files results in monotonically increasing output.
- *Update:* The engine updates the sink in place based on a key for each record, updating only keys whose values changed.

Figure 2 illustrates the model visually. One attractive property of the model is that the *contents* of the result table (which is logically just a view that need never be materialized) are defined independently of the output mode (whether we output the whole table on every trigger, or only deltas). In contrast, APIs such as Dataflow require the equivalent of an output mode on every operator, so users must plan the whole operator DAG keeping in mind whether each operator is outputting complete results or positive or negative deltas, effectively incrementalizing the query by hand.

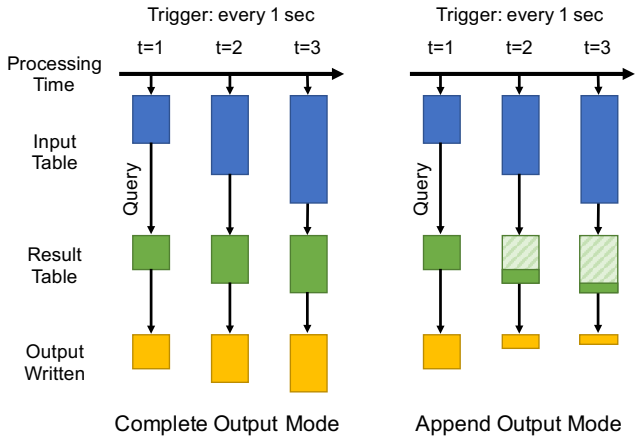


Figure 2: Structured Streaming's semantics for two output modes. Logically, all input data received up to a point in processing time is viewed as a large input table, and the user provides a query that defines a result table based on this input. Physically, Structured Streaming computes changes to the result table *incrementally* (without having to store all input data) and outputs results based on its output mode. For complete mode, it outputs the whole result table (left), while for append mode, it only outputs newly added records (right).

A second attractive property is that the model has strong consistency semantics, which we call *prefix consistency*. First, it guarantees that when input records are relatively ordered within a source (e.g., log records from the same device), the system will only produce results that incorporate them in the same records (e.g., never skipping a record). Second, because the result table is defined based on *all* data in the input prefix at once, we know that all rows in the result table reflect all input records. In contrast, in some systems based on message-passing between nodes, the node that receives a record might send an update to two downstream nodes, but there is no guarantee that the outputs from these two are synchronized. Prefix consistency also makes operation easier, as users can roll back the system to a specific point in the write-ahead log (i.e., a specific prefix of the data) and recompute outputs from that point.

In summary, with the Structured Streaming models, as long as users understand a regular Spark or DataFrame query, they can understand the content of the result table for their job and the values that will be written to the sink. Users need not worry about consistency, failures or incorrect processing orders.

Finally, the reader might notice that some of the output modes we defined are incompatible with certain types of query. For example, suppose we are aggregating counts by country, as in our code example in the previous section, and we want to use the append output mode. There is no way for the system to guarantee it has stopped receiving records for a given country, so this combination of query and output mode will not be allowed by the system. We describe which combinations are allowed in Section 5.1.

4.3 Streaming Specific Operators

Many Structured Streaming queries can be written using just the standard operators in Spark SQL, such as selection, aggregation and

joins. However, to support some requirements unique to streaming, we added two new types of operators to Spark SQL: *watermarking* operators tell the system when to "close" an event time window and output results or forget state, and *stateful operators* let users write custom logic to implement complex processing. Crucially, both of these new operators still fit in Structured Streaming's incremental semantics (§4.2), and both can also be used in batch jobs.

4.3.1 Event Time Watermarks. From a logical point of view, the key idea in event time is to treat application-specified timestamps as an arbitrary field in the data, allowing records to arrive out-of-order [2, 24]. We can then use standard operators and incremental processing to update results grouped by event time. In practice, however, it is useful for the processing system to have some loose bounds on how late data can arrive, for two reasons:

(1) Allowing arbitrarily late data might require storing arbitrarily large state. For example, if we count data by 1-minute event time window, the system needs to remember a count for every 1-minute window since the application began, because a late record might still arrive for any particular minute. This can quickly lead to large amounts of state, especially if combined with another grouping key. The same issue happens with joins.

(2) Some sinks do not support data retraction, making it useful to be able to write the results for a given event time after a timeout. For example, custom downstream applications want to start working with a "final" result and might not support retractions. Append-mode sinks also do not support retractions.

Structured Streaming lets developers set a watermark [2] for event time columns using the `withWatermark` operator. This operator gives the system a delay threshold t_C for a given timestamp column C. At any point in time, the watermark for C is $\max(C) - t_C$, that is, t_C seconds before the maximum event time seen so far in C. Note that this choice of watermark is naturally robust to backlogged data: if the system cannot keep up with the input rate for a period of time, the watermark will not move forward arbitrarily during that time, and all events that arrived within at most T seconds of being produced will still be processed.

When present, watermarks affect when stateful operators can forget old state (e.g., if grouping by a window derived from a watermarked column), and when Structured Streaming will output data with an event time key to append-mode sinks. Different input streams can have different watermarks.

4.3.2 Stateful Operators. For developers who want to write custom stream processing logic, Structured Streaming's stateful operators are "UDFs with state" that give users control over the computation while fitting into Structured Streaming's semantics and fault tolerance mechanisms. There are two stateful operators, `mapGroupsWithState` and `flatMapGroupsWithState`. Both operators act on data that has been assigned a *key* using `groupByKey`, and let the developers track and update a *state* for each key using custom logic, as well as output records for each key. They are closely based on Spark Streaming's `updateStateByKey` operator [37].

The `mapGroupsWithState` operator, on a grouped dataset with keys of type K and values of type V, takes in a user-defined *update function* with the following arguments:

- `key` of type K
- `newValues` of type Iterator[V]
- `state` of type GroupState[S], where S is a user-specified class.

```
// Define an update function that simply tracks the
// number of events for each key as its state, returns
// that as its result, and times out keys after 30 min.
def updateFunc(key: UserId, newValues: Iterator[Event],
               state: GroupState[Int]): Int = {
  val totalEvents = state.get() + newValues.size()
  state.update(totalEvents)
  state.setTimeoutDuration("30 min")
  return totalEvents
}

// Use this update function on a stream, returning a
// new table lens that contains the session lengths.
lens = events.groupByKey(event => event.userId)
             .mapGroupsWithState(updateFunc)
```

Figure 3: Using `mapGroupsWithState` to track the number of events per session, timing out sessions after 30 minutes.

The operator will invoke this function whenever one or more new values are received for a key. On each call, the function receives all of the values that were received for that key since the last call (multiple values may be batched for efficiency). It also receives a `state` object that wraps around a user-defined data type S, and allows the user to update the state, drop this key from state tracking, or set a timeout for this specific key (either in event time or processing time). This allows the user to store arbitrary data for the key, as well as implement custom logic for dropping state (e.g., custom exit conditions when implementing session-based windows).

Finally, the update function returns a user-specified return type R for its key. The return value of `mapGroupsWithState` is a new table with the final R record outputted for each group in the data (when the group is closed or times out). For example, the developer may wish to track user sessions on a website using `mapGroupsWithState`, and output the total number of pages clicked for each session.

To illustrate, Figure 3 shows how to use `mapGroupsWithState` to track user sessions, where a session is defined as a series of events with the same `userId` and gaps less than 30 minutes between them. We output the final number of events in each session as our return value R. A job could then compute metrics such as the average number of events per session by aggregating the result table `lens`.

The second stateful operator, `flatMapGroupsWithState`, is very similar to `mapGroupsWithState`, except that the update function can return *zero or more* values of type R per update instead of one. For example, this operator could be used to manually implement a stream-to-table join. The return values can either be returned all at once, when the group is closed, or incrementally across calls to the update function. Both operators also work in batch mode, in which case the update function will only be called once.

5 Query Planning

We implemented Structured Streaming's query planning using the Catalyst extensible optimizer in Spark SQL [8], which allows writing composable rules using pattern matching in Scala. Query planning proceeds in three stages: analysis to determine whether the query is valid, incrementalization and optimization.

5.1 Analysis

The first stage of query planning is analysis, where the engine validates the user's query and resolves the attributes and data types referred to in the query. Structured Streaming uses Spark SQL's existing analysis passes to resolve attributes and types, but adds new rules to check that the query can be executed incrementally by the engine. It also checks that the user's chosen output mode is valid for this specific query. For example, the Append output mode can only be used with queries whose output is monotonic [4]: that is, where a given output record will not be removed once it is written. In this mode, only selections, joins, and aggregations over keys that include event time are allowed (in which case the engine will only output the value for a given event time once its watermark has passed). Similarly, in the Complete output mode, where the whole output table needs to be written on each trigger, Structured Streaming only permits aggregation queries where the amount of state that needs to be tracked is proportional to the number of keys in the result. A full description of the supported modes is available in the Structured Streaming documentation [31].

5.2 Incrementalization

The next step of the query planning process is incrementalizing the static query provided by the user to efficiently update results in response to new data. In general, Structured Streaming's incrementalizer aims to ensure that the query's result can be updated in time proportional to the amount of new data received before each trigger or to the amount of new rows that have to be produced, without a dependance on the total amount of data received so far.

The engine can incrementalize a restricted, but growing, class of queries. As of Spark 2.3.0, the supported queries can contain:

- Any number of selections, projections and `SELECT DISTINCT`s.
- Inner, left-outer and right-outer joins between a stream and a table or between two streams. For outer joins against a stream, the join condition must involve a watermarked column.
- Stateful operators like `mapGroupsWithState` (§4.3.2).
- Up to one aggregation (possibly on compound keys).
- Sorting after an aggregation, only in complete output mode.

The engine uses Catalyst transformation rules to map these supported queries into trees of physical operators that perform both computation and state management. For example, an aggregation in the user query might be mapped to a StatefulAggregate operator that tracks open groups inside Structured Streaming's state store (§6.1) and outputs the desired result. Internally, Structured Streaming also tracks an output mode for each physical operator in the DAG produced during incrementalization, similar to the refinement mode for aggregation operators in Dataflow [2]. For example, some operators may update emitted records (equivalent to update mode), while others may only emit new records (append mode). Crucially, in Structured Streaming, users do not have to specify these intra-DAG modes manually.

Incrementalization is an active area of work in Structured Streaming, but we have found that even the restricted set of queries available today is suitable for many use cases (§8). In other cases, users have leveraged Structured Streaming's stateful operators (§4.3.2) to implement custom incremental processing logic that maintains state of their choice. We expect to add more advanced automatic incrementalization techniques into the engine over time.

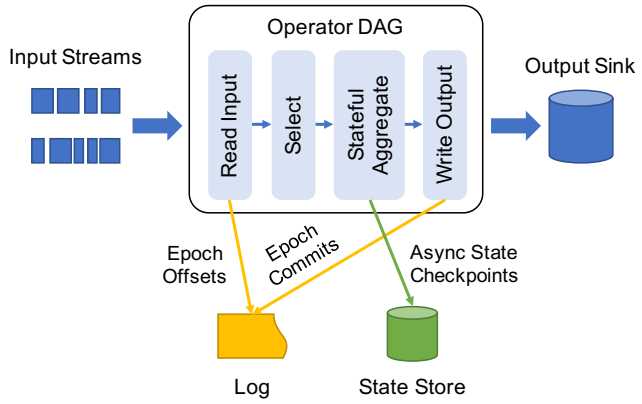


Figure 4: State management during the execution of Structured Streaming. Input operators are responsible for defining epochs in each input source and saving information about them (e.g., offsets) reliably in the write-ahead log. Stateful operators also checkpoint state asynchronously, marking it with its epoch, but this does not need to happen on every epoch. Finally, output operators log which epochs' outputs have been reliably committed to the idempotent output sink; the very last epoch may be rewritten on failure.

5.3 Query Optimization

The final stage of planning is optimization. Structured Streaming applies most of the optimization rules in Spark SQL [8], such as predicate pushdown, projection pushdown, expression simplification and others. In addition, it uses Spark SQL's Tungsten binary format for data in memory (avoiding the overhead of Java objects), and its runtime code generator to compile chains of operators to Java bytecode that runs over this format. This design means that most of the work in logical and execution optimization for analytical workloads in Spark SQL automatically applies to streaming.

6 Application Execution

The final component of Structured Streaming is its execution strategy. In this section, we describe how the engine tracks state, and then the two execution modes: microbatching via fine-grained tasks and continuous processing using long-lived operators. We then discuss operational features to simplify managing and deploying Structured Streaming applications.

6.1 State Management and Recovery

At a high level, Structured Streaming tracks state in a manner similar to Spark Streaming [37], in both its microbatch and continuous modes. The state of an application is tracked using two external storage systems: a *write-ahead log* that supports durable, atomic writes at low latency, and a *state store* that can store larger amounts of data durably and allows parallel access (e.g., S3 or HDFS). Structured Streaming uses these systems together to recover on failure.

The engine places two requirements on input sources and output sinks to provide fault tolerance. First, input sources should be *replayable*, i.e., allow re-reading recent data using some form of

identifier, such as a stream offset. Durable message bus systems like Kafka and Kinesis meet this need. Second, output sinks should be *idempotent*, allowing Structured Streaming to rewrite some already written data on failure. Sinks can implement this in different ways.

Given these properties, Structured Streaming performs state tracking using the following mechanism, as shown in Figure 4:

(1) As input operators read data, the master node of the Spark application defines *epochs* based on offsets in each input source. For example, Kafka and Kinesis present topics as a series of partitions, each of which are byte streams, and allow reading data using offsets in these partitions. The master writes the start and end offsets of each epoch durably to the log.

(2) Any operators requiring state checkpoint their state periodically and asynchronosuly to the state store, using incremental checkpoints when possible. They store the epoch ID along with each checkpoint written. These checkpoints do not need to happen on every epoch or to block processing.[2]

(3) Output operators write the epochs they committed to the log. The master waits for all nodes running an operator to report a commit for a given epoch before allowing commits for the next epoch. Depending on the sink, the master can also run an operation to finalize the writes from multiple nodes if the sink supports this. This means that if the streaming application fails, only one epoch may be partially written.[3]

(4) Upon recovery, the new instance of the application starts by reading the log to find the last epoch that has not been committed to the sink, including its start and end offsets. It then uses the offsets of earlier epochs to reconstruct the application's in-memory state from the last epoch written to the state store. This just requires loading the old state and running those epochs with the same offsets while disabling output. Finally, the system reruns the last epoch and relies on the sink's idempotence to write its results, then starts defining new epochs.

Finally, all of the state management in this design is transparent to user code. Both the aggregation operators and custom stateful processing operators (e.g., `mapGroupsWithState`) automatically checkpoint state to the state store, without requiring custom code to do it. The user's data types only need to be serializable.

6.2 Microbatch Execution Mode

Structured Streaming jobs can execute in two modes: microbatching or continuous operators. The microbatch mode uses the *discretized streams* execution model from Spark Streaming [37], and inherits its benefits, such as dynamic load balancing, rescaling, straggler mitigation and fault recovery without whole-system rollback.

In this mode, epochs are typically set to be a few hundred milliseconds to a few seconds, and each epoch executes as a traditional Spark job composed of a DAG of independent tasks [36]. For example, a query doing selection followed by stateful aggregation might execute as a set of "map" tasks for the selection and "reduce" tasks for the aggregation, where the reduce tasks track state in memory on worker nodes and periodically checkpoint it to the state store. As in Spark Streaming, this mode provides the following benefits:

- **Dynamic load balancing:** Each operator's work is divided into small, independent tasks that can be scheduled on any node, so the system can automatically balance these across nodes if some are executing slower than others.
- **Fine-grained fault recovery:** If a node fails, only its tasks need to be rerun, instead of having to roll back the whole cluster to a checkpoint as in most systems based on topologies of long-lived operators. Moreover, the lost tasks can be rerun *in parallel*, further reducing recovery time [37].
- **Straggler mitigation:** Spark will launch backup copies of slow tasks as it does in batch jobs, and downstream tasks will simply use the output from whichever copy finishes first.
- **Rescaling:** Adding or removing a node is simple as tasks will automatically be scheduled on all the available nodes.
- **Scale and throughput:** Because this mode reuses Spark's batch execution engine, it inherits all the optimizations in this engine, such as a high-performance shuffle implementation [34] and the ability to run on thousands of nodes.

The main disadvantage of this mode is a higher minimum latency, as there is overhead to launching a DAG of tasks in Spark. In practice, however, latencies of a few seconds are achievable even on large clusters running multi-step computations. Depending on the application, these are on a similar time scale to data collection and alerting systems.

6.3 Continuous Processing Mode

A new continuous processing added in Apache Spark 2.3 [6] executes Structured Streaming jobs using long-lived operators as in traditional streaming systems such as Telegraph and Borealis [1, 13]. This mode enables lower latency at a cost of less operational flexibility (e.g., limited support for rescaling the job at runtime).

The key enabler for this execution mode was choosing a declarative API for Structured Streaming that is not tied to the execution strategy. For example, the original Spark Streaming API had some operators based on processing time that leaked the concept of microbatches into the programming model, making it hard to move programs to another type of engine. In contrast, Structured Streaming's API and semantics are independent of the execution engine: continuous execution is similar to having a much larger number of triggers. Note that unlike systems based purely on unsynchronized message passing, such as Storm [32], we do retain the concept of triggers and epochs in this mode so the output from multiple nodes can be coordinated and committed together to the sink.

Because the API supports fine-grained execution, Structured Streaming jobs could theoretically run on any existing distributed streaming engine design [1, 13, 17]. In continuous processing, we built a simple continuous operator engine that lives inside Spark and can reuse Spark's scheduling infrastructure and per-node operators (e.g., code-generated operators). The first version released in Spark 2.3.0 only supports "map-like" jobs (i.e., no shuffle operations), which were one of the most common scenarios where users wanted lower latency, but the design can be extended to support shuffles.

[2] In Spark 2.3.0, we actually make one checkpoint per epoch, but we plan to make them less frequent in a future release, as is already done in Spark Streaming.

[3] Some sinks, such as Amazon S3, provide no way to atomically commit multiple writes from different writer nodes. In such cases, we have also created Spark data sources that add transactions over the underlying storage system. For example, Databricks Delta [7] offers a consistent view of S3 data for both streaming and batch queries, along with additional features such as index maintenance.

Compared to microbatch execution, there are two differences when using continuous processing:

(1) The master launches long-running tasks on each partition using Spark's scheduler that each read one partition of the input source (e.g., Kinesis stream) but execute *multiple* epochs. If one of these tasks fails, Spark will simply relaunch it.

(2) Epochs are coordinated differently. The master periodically tells nodes to start a new epoch, and receives a start offset for the epoch on each input partition, which it inserts into the write-ahead log. When it asks them to start the next epoch, it also receives end offsets for the previous one, writes these to the log, and tells nodes to commit the epoch when it has written all the end offsets. Thus, the master is not on the critical path for inspecting all the input sources and defining start/end offsets.

We found that the most common use case where organizations wanted low latency *and* the scale of a distributed processing engine was "stream to stream" map operations to transform data before it is used in other streaming applications. For example, an organization might upload events to Kafka, run some simple ETL transformations as a streaming job, and write the transformed data to Kafka again for consumption by other streaming applications. In this type of design, each transformation job will add latency to all downstream steps, so organizations wish to minimize this latency.

7 Operational Features

We used several properties of our execution strategy and API to design a number of operational features in Structured Streaming that tackle common problems in deployments. Perhaps most importantly across these features, we aimed to make both Structured Streaming's semantics and its fault tolerance model easy to understand. With a simple design, operators can form an accurate model of how a system runs and what various actions will do to it.

7.1 Code Updates

Developers can update User-Defined Functions (UDFs) in their program and simply restart the application to use the new version of the code. For example, if a UDF is crashing on a particular input record, that epoch of processing will fail, so the developer can update the code and restart the application again to continue processing. This also applies to stateful operator UDFs, which can be updated as long as they retain the same schema for their state objects. We also designed Spark's log and state store formats to be binary compatible across Spark framework updates.

7.2 Manual Rollback

Sometimes, an application outputs *wrong* results for some time before a user notices: for example, a field that fails to parse might simply be reported as `NULL`. Therefore, rollbacks are a fact of life for many operators. In Structured Streaming, it is easy to determine which records went into each epoch from the write-ahead log and roll back the application to the epoch where a problem started occurring. We chose to store the write-ahead log as JSON to let administrators perform these operations manually.[4] As long as the input sources and state store still have data from the failed epoch, the job can start again from a previous point. Message buses like Kafka are typically configured for several weeks of retention so rollbacks are often possible.

Manual rollbacks interact well with Structured Streaming's prefix consistency guarantee for execution semantics 4.2. Specifically, when an administrator rolls back the job to a point in the write-ahead log, she knows which prefix of the input streams this point corresponds to, and the job can recompute output from that point on while retaining consistency within the new output. Beyond this guarantee, Structured Streaming's support for running the same code as a batch job and for rescaling means that administrators can run the recovery on a temporarily larger cluster to catch up quickly, further reducing the operational complexity of manual rollbacks.

7.3 Hybrid Batch and Streaming Execution

The most obvious benefit of Structured Streaming's unified API is that users can share code between batch and streaming jobs, or run the same program as a batch job for testing. However, we have also found this useful for purely streaming scenarios in two ways:

- *"Run-once" triggers for cost savings:* Many Databricks customers wanted the transactionality and state management properties of a streaming engine *without* running servers 24/7. Virtually all ETL workloads require tracking how far in the input one has gotten and which results have been saved reliably, which can be difficult to implement by hand. These functions are exactly what Structured Streaming's state management provides. Thus, several customers implemented ETL jobs by running a *single* epoch of a Structured Streaming job every few hours as a batch computation, using the provided "run once" trigger that was originally designed for testing. This leads to significant cost savings (in one case, up to 10× [35]) for lower-volume applications. With all the major cloud providers now supporting per-second or per-minute billing [9], we believe this type of "discontinuous processing" will become more common.
- *Adaptive batching:* Even streaming applications occasionally experience large backlogs. For example, a link between two datacenters might go down, temporarily delaying data transfer, or there might simply be a spike in user activity. In these cases, Structured Streaming will automatically execute longer epochs in order to catch up with the input streams, often achieving similar throughput to Spark's batch jobs. This will not greatly increase latency, given that data is already backlogged, but will let the system catch up faster. In cloud environments, operators can also add extra nodes to the cluster temporarily.

7.4 Monitoring

Structured Streaming uses Spark's existing metrics API and structured event log to report information such as number of records processed, bytes shuffled across the network, etc. These interfaces are familiar to operators and easy to connect to a variety of UI tools.

7.5 Fault and Straggler Recovery

As discussed in §6.2, Structured Streaming's microbatch mode can recover from node failures, stragglers and load imbalances using Spark's fine-grained task execution model. The continuous processing mode recovers from node failures, but does not yet protect against stragglers or load imbalance.

[4] One additional step they may have to do is remove faulty data from the output sink, depending on the sink chosen. For the file sink, for example, it's straightforward to find which files were written in a particular epoch and remove those.

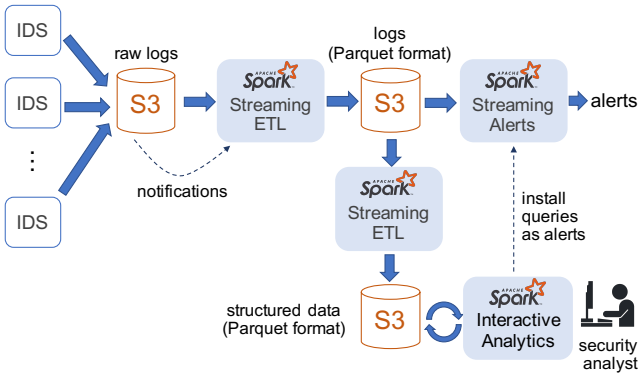


Figure 5: Information security platform use case. Using Structured Streaming and Spark SQL, a team of analysts can query both streaming and historical data and easily install queries for new attack patterns as streaming alerts.

8 Production Use Cases

We have supported Structured Streaming on Databricks' managed cloud service [16] since 2016, and today, our cloud is running hundreds of production streaming applications at a given time (i.e., applications running 24/7). The largest of these applications ingest over 1 PB of data per month and run on hundreds of servers. We also use Structured Streaming internally to monitor our services, including the execution of Structured Streaming itself. In this section, we describe three customer workloads that leverage various aspects of Structured Streaming, as well as our internal use case.

8.1 Information Security Platform

A large customer has used Structured Streaming to develop a large-scale security platform to enable over 100 analysts to scour through network traffic logs to quickly identify and respond to security incidents, as well as to generate automated alerts. This platform combines streaming with batch and interactive queries and is thus a great example of the system's support for *end-to-end* applications.

Figure 5 shows the architecture of the platform. Intrusion Detection Systems (IDSes) monitor all the network traffic in the organization, and output logs to S3. From here, a Structured Streaming jobs ETLs these logs into a compact Apache Parquet based table stored on Databricks Delta [7] to enable fast and concurrent access from multiple downstream applications. Other Structured Streaming jobs then process these logs to produce additional tables (e.g., by joining them with other data). Analysts query these tables interactively, using SQL or Dataframes, to detect and diagnose new attack patterns. If they identify a compromise, they also look back through historical data to trace previous actions from that attacker. Finally, in parallel, the Parquet logs are processed by another Structured Streaming cluster that generates real-time alerts based on pre-written rules.

The key challenges in realizing this platform are (1) building a robust and scalable streaming pipeline, while (2) providing the analysts with an effective environment to query both fresh and historical data. Using standard tools and services available on AWS, a team of 20 people took over six months to build and deploy a previous version of this platform in production. This previous version had several limitations, including only being able to store a small amount of data for historical queries due to using a traditional data warehouse for the interactive queries. In contrast, a team of five engineers was able to reimplement the platform using Structured Streaming in two weeks. The new platform was simultaneously more scalable and able to support more complex analysis using Spark's ML APIs. Next, we provide a few examples to illustrate the advantages of Structured Streaming that made this possible.

First, Structured Streaming's ability to adaptively vary the batch size enabled the developers to build a streaming pipeline that deals not only with spikes in the workload, but also with failures and code upgrades. Consider a streaming job that goes offline either due to failure or upgrades. When the cluster is brought back online, it will start automatically to process the data all the way back from the moment it went offline. Initially, the cluster will use large batches to maximize the throughput. Once it catches up, the cluster switches to small batches for low latency. This allows administrators to regularly upgrade clusters without the fear of excessive downtime.

Second, the ability to join a stream with other streams, as well as with historical tables, has considerably simplified the analysis. Consider the simple task of figuring out which device a TCP connection originates at. It turns out that this task is challenging in the presence of mobile devices, as these devices are given *dynamic* IP addresses every time they join the network. Hence, from TCP logs alone, is not possible to track down the end-points of a connection. With Structured Streaming, an analyst can easily solve this problem. She can simply join the TCP logs with DHCP logs to map the IP address to the MAC address, and then use the organization's internal database of network devices to map the MAC address to a particular machine and user. In addition, users were able to do this join in real time using stateful operators as both the TCP and DHCP logs were being streamed in.

Finally, using the same system for streaming, interactive queries and ETL has provided developers with the ability to quickly iterate and deploy new alerts. In particular, it enables analysts to build and test queries for detecting new attacks on offline data, and then deploy these queries directly on the alerting cluster. In one example, an analyst developed a query to identify exfiltration attacks via DNS. In this attack, malware leaks confidential information from the compromised host by piggybacking this information into DNS requests sent to an external DNS server owned by the attacker. One simplified query to detect such an attack essentially computes the aggregate size of the DNS requests sent by every host over a time interval. If the aggregate is greater than a given threshold, the query flags the corresponding host as potentially being compromised. The analyst used historical data to set this threshold, so as to achieve the desired balance between false positive and false negative rates. Once satisfied with the result, the analyst simply pushed the query to the alerting cluster. The ability to use the same system and the same API for data analysis and for implementing the alerts led not only to significant engineering cost savings, but also to better security, as it is significantly easier to deploy new rules.

8.2 Monitoring Live Video Delivery

A large media company is using Structured Streaming to compute quality metrics for their live video traffic and interactively identify delivery problems. Live video delivery is especially challenging

because network problems can severely disrupt utility. For pre-recorded video, clients can use large buffers to mask issues, and a degradation at most results in extra buffering time; but for live video, a problem may mean missing a critical moment in a sports match or similar event. This organization collects video quality metrics from clients in real time, performs ETL operations and aggregation using Structured Streaming, then stores the results in a data warehouse. This allows operations engineers to interactively query fresh data to detect and diagnose quality issues (e.g., determine whether an issue is tied to a specific ISP, video server or other cause).

8.3 Analyzing Game Performance

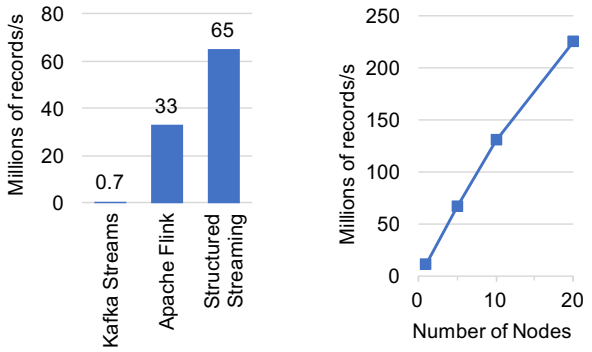
A large gaming company uses Structured Streaming to monitor the latency experienced by players in a popular online game with tens of millions of monthly active users. As in the video use case, high network performance is essential for the user experience when gaming, and repeated problems can quickly lead to player churn. This organization collects latency logs from its game clients to cloud storage and then performs a variety of streaming analyses. For example, one job joins the measurements with a table of Internet Autonomous Systems (ASes) and then aggregates the performance by AS over time to identify poorly performing ASes. When such an AS is identified, the streaming job triggers an alert, and IT staff can contact the AS in question to remediate the issue.

8.4 Cloud Monitoring at Databricks

At Databricks, we have been using Apache Spark since the start of the company to monitor our own cloud service, understand workload statistics, trigger alerts, and let our engineers interactively debug issues. The monitoring pipeline produces dozens of interactive dashboards as well as structured Parquet tables for ad-hoc SQL queries. These dashboards also play a key role for business users to understand which customers have increasing or decreasing usage, prioritize feature development, and proactively identify customers that are experiencing problems.

We built at least three versions of a monitoring pipeline using a combination of batch and streaming APIs starting four years ago, and in all the cases, we found that the major challenges were operational. Despite our best efforts, pipelines could be brittle, experiencing frequent failures when aspects of our input data changed (e.g., new schemas or reading from more locations than before), and upgrading them was a daunting exercise. Worse yet, failures and upgrades often resulted in missing data, so we had to manually go back and re-run jobs to reconstruct the missing data. Testing pipelines was also challenging due to their reliance on multiple distinct Spark jobs and storage systems. Our experience with Structured Streaming shows that it successfully addresses many of these challenges. Not only we were able to reimplement our pipelines in weeks, but the management overhead decreased drastically. Restartability coupled with adaptive batching, transactional sources/sinks and well-defined consistency semantics have enabled simpler fault recovery, upgrades, and rollbacks to repair old results. Moreover, we can test the same code in batch mode on data samples or use many of the same functions in interactive queries.

Our pipelines with Structured Streaming also regularly combine its batch and streaming capabilities. For example, the pipeline to monitor streaming jobs starts with an ETL job that reads JSON



(a) vs. Other Systems (b) System Scaling

Figure 6: Throughput results on the Yahoo! benchmark.

events from Kafka and writes them to a columnar Parquet table in S3. Dozens of other batch and streaming jobs then query this table to produce dashboards and other reports. Because Parquet is a compact and column-oriented format, this architecture consumes drastically *fewer* resources than having every job read directly from Kafka, and simultaneously places less load on the Kafka brokers. Overall, streaming jobs' latencies range from seconds to minutes, and users can also query the Parquet table interactively in seconds.

9 Performance Evaluation

In this section, we measure the performance of Structured Streaming using controlled benchmarks. We study performance vs. other systems on the Yahoo! Streaming Benchmark [14], scalability, and the throughput-latency tradeoff with continuous processing.

9.1 Performance vs. Other Streaming Systems

To evaluate performance compared to other streaming engines, we used the Yahoo! Streaming Benchmark [14], a widely used workload that has also been evaluated in other open source systems. This benchmark requires systems to read ad click events, join them against a static table of ad campaigns by campaign ID, and output counts by campaign on 10-second event-time windows.

We compared Kafka Streams 0.10.2, Apache Flink 1.2.1 and Spark 2.3.0 on a cluster with five `c3.2xlarge` Amazon EC2 workers (each with 8 virtual cores and 15 GB RAM) and one master. For Flink, we used the optimized version of the benchmark published by dataArtisans for a similar cluster [22]. Like in that benchmark, the systems read data from a Kafka cluster running on the workers with 40 partitions (one per core), and write results to Kafka. The original Yahoo! benchmark used Redis to hold the static table for joining ad campaigns, but we found that Redis could be a bottleneck, so we replaced it with a table in each system (a `KTable` in Kafka, a DataFrame in Spark, and an in-memory hash map in Flink).

Figure 6a shows each system's maximum stable throughput, i.e., the throughput it can process before a backlog begins to form. We see that streaming system performance can vary significantly. Kafka Streams implements a simple message-passing model through the Kafka message bus, but only attains 700,000 records/second on our 40-core cluster. Apache Flink reaches 33 million records/s. Finally, Structured Streaming reaches 65 million records/s, nearly 2× the throughput of Flink. This particular Structured Streaming query is

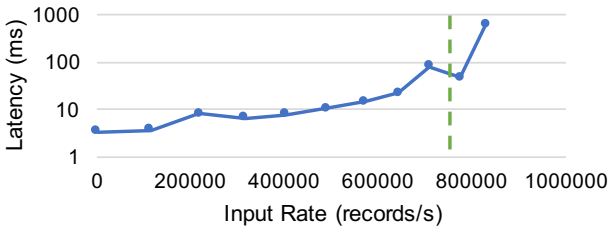


Figure 7: Latency of continuous processing vs. input rate. Dashed line shows max throughput in microbatch mode.

implemented using just DataFrame operations with no UDF code. The performance thus comes solely from Spark SQL's built in execution optimizations, including storing data in a compact binary format and runtime code generation. As pointed out by the authors of Trill [12] and others, execution optimizations can make a large difference in streaming workloads, and many systems based on per-record operations do not maximize performance.

9.2 Scalability

Figure 6b shows how Structured Streaming's performance scales for the Yahoo! benchmark as we vary the size of our cluster. We used 1, 5, 10 and 20 `c3.2xlarge` Amazon EC2 workers (with 8 virtual cores and 15 GB RAM each) and the same experimental setup as in §9.1, including one Kafka partition per core. We see that throughput scales close to linearly, from 11.5 million records/s on 1 node to 225 million records/s on 20 nodes (i.e., 160 cores).

9.3 Continuous Processing

We benchmarked Structurd Streaming's continuous processing mode on a 4-core server to show the latency-throughput tradeoffs it can achieve. (Because partitions run independently in this mode, we expect the latency to stay the same as more nodes are added.) Figure 7 shows the results for a `map` job reading from Kafka, with the dashed line showing the maximum throughput achievable by microbatch mode. We see that continuous mode is able to achieve much lower latency without a large drop in throughput (e.g., less than 10 ms latency at half the maximum throughput of microbatching). Its maximum stable throughput is also slightly higher because microbatch mode incurs latency due to task scheduling.

10 Related Work

Structured Streaming builds on many existing systems for stream processing and big data analytics, including Spark SQL's DataFrame API [8], Spark Streaming [37], Dataflow [2], incremental query systems [11, 24, 29, 38] and distributed stream processing [21]. At a high level, the main contributions of this work are:

- An account of real-world user challenges with streaming systems, including operational challenges that are not always discussed in the research literature (§2).
- A simple, declarative programming model that *incrementalizes* a widely used batch API (Spark DataFrames/SQL) to provide similar capabilities to Dataflow [2] and other streaming systems.
- An execution engine providing high throughput, fault tolerance, and rich operational features that combines with the rest of Apache Spark to let users easily build *end-to-end* applications.

From an API standpoint, the closest work is incremental query systems [11, 24, 29, 38], including recent distributed systems such as Stateful Bulk Processing [25] and Naiad [26]. Structured Streaming's API is an extension of Spark SQL [8], including its declarative DataFrame interface for programmatic construction of relational queries. Apache Flink also recently added a table API (currently in beta) for defining relational queries that can map to either streaming or batch execution [19], but this API lacks some of the features of Structured Streaming, such as custom stateful operators (§4.3.2).

Other recent streaming systems have language-integrated APIs that operate at a lower, more "imperative" level. In particular, Spark Streaming [37], Google Dataflow [2] and Flink's DataStream API [18] provide various functional operators but require users to choose the right DAG of operators to implement a particular incrementalization strategy (e.g., when to pass on deltas versus complete results); essentially, these are equivalent to writing a physical execution plan. Structured Streaming's API is simpler for users who are not experts on incrementalization. Structured Streaming adopts the definitions of event time, processing time, watermarks and triggers from Dataflow but incorporates them in an incremental model.

For execution, Structured Streaming uses concepts similar to discretized streams for microbatch mode [37] and traditional streaming engines for continuous processing mode [1, 13, 21]. It also builds on an analytical engine for performance like Trill [12]. The most unique contribution here is the integration of batch and streaming queries to enable sophisticated end-to-end applications. As described in §8, Structured Streaming users can easily write applications that combine batch, interactive and stream processing using the same code (e.g., security log analysis). In addition, they leverage powerful operational features such as run-once triggers (running a streaming application "discontinuously" as batch jobs to retain its transactional features but lower costs), code updates, and batch processing to handle backlogs or code rollbacks (§7).

11 Conclusion

Stream processing is a powerful tool, but streaming systems are still difficult to use, operate and integrate into larger applications. We designed Structured Streaming to simplify all three of these tasks while integrating with the rest of Apache Spark. Unlike many other open source streaming engines, Structured Streaming purposefully adopts a very high-level API: incrementalizing an existing Spark SQL or DataFrame query. This makes it accessible to a wide range of users. Although Structured Streaming's API is more declarative and constrained, we found that works well for a diverse range of applications, including those that require custom logic for stateful processing. Beyond this focus on a high-level API, Structured Streaming also includes several powerful operational features and achieves high performance using the Spark SQL engine. Experience across hundreds of customer use cases shows that users can leverage the system to build sophisticated business applications.

12 Acknowledgements

We would like to thank the diverse Apache Spark developer community that has contributed to Structured Streaming, Spark Streaming and Spark SQL over the years. We also thank the SIGMOD reviewers for their detailed feedback on the paper.

References

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. 2005. The design of the borealis stream processing engine. In *In CIDR*. 277–289.

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 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 (Aug. 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[3] Intel Altera. 2017. Financial/HPC – Financial Offload. https://www.altera.com/solutions/industry/computer-and-storage/applications/computer/financial-offload.html. (2017).

[4] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency analysis in Bloom: A CALM and collected approach. In *In Proceedings 5th Biennial Conference on Innovative Data Systems Research*. 249–260.

[5] Amazon. 2017. Amazon Kinesis. https://aws.amazon.com/kinesis/. (2017).

[6] Michael Armbrust. 2017. SPARK-20928: Continuous Processing Mode for Structured Streaming. https://issues.apache.org/jira/browse/SPARK-20928. (2017).

[7] Michael Armbrust, Bill Chambers, and Matei Zaharia. 2017. Databricks Delta: A Unified Data Management System for Real-time Big Data. https://databricks.com/blog/2017/10/25/databricks-delta-a-unified-management-system-for-real-time-big-data.html. (2017).

[8] 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. 1383–1394. https://doi.org/10.1145/2723372.2742797

[9] Jeff Barr. 2017. New – Per-Second Billing for EC2 Instances and EBS Volumes. https://aws.amazon.com/blogs/aws/new-per-second-billing-for-ec2-instances-and-ebs-volumes/. (2017).

[10] Apache Beam. 2017. Apache Beam programming guide. https://beam.apache.org/documentation/programming-guide/. (2017).

[11] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. 1986. Efficiently Updating Materialized Views. *SIGMOD Rec.* 15, 2 (June 1986), 61–71. https://doi.org/10.1145/16856.16861

[12] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C. Platt, James F. Terwilliger, and John Wernsing. 2014. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 401–412. https://doi.org/10.14778/2735496.2735503

[13] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*. ACM, New York, NY, USA, 668–668. https://doi.org/10.1145/872757.872857

[14] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Tom Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Peng, and Paul Poulosky. 2015. Benchmarking Streaming Computation Engines at Yahoo! https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at. (2015).

[15] Confluent. 2017. KSQL: Streaming SQL for Kafka. https://www.confluent.io/product/ksql/. (2017).

[16] Databricks. 2017. Databricks unified analytics platform. https://databricks.com/product/unified-analytics-platform. (2017).

[17] Apache Flink. 2017. Apache Flink. http://flink.apache.org. (2017).

[18] Apache Flink. 2017. Flink DataStream API Programming Guide. https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/datastream_api.html. (2017).

[19] Apache Flink. 2017. Flink Table & SQL API Beta. https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/table/index.html. (2017).

[20] Apache Flink. 2017. Working with State. https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/stream/state.html. (2017).

[21] Lukasz Golab and M. Tamer Özsu. 2003. Issues in Data Stream Management. *SIGMOD Rec.* 32, 2 (June 2003), 5–14. https://doi.org/10.1145/776985.776986

[22] Jamie Grier. 2016. Extending the Yahoo! Streaming Benchmark. https://data-artisans.com/blog/extending-the-yahoo-streaming-benchmark. (2016).

[23] Apache Kafka. 2017. Kafka. http://kafka.apache.org. (2017).

[24] Sailesh Krishnamurthy, Michael J. Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. 2010. Continuous Analytics over Discontinuous Streams. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 1081–1092. https://doi.org/10.1145/1807167.1807290

[25] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. 2010. Stateful Bulk Processing for Incremental Analytics. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1807128.1807138

[26] Frank McSherry, Derek Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential dataflow, In Proceedings of CIDR 2013. https://www.microsoft.com/en-us/research/publication/differential-dataflow/

[27] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. 439–455. https://doi.org/10.1145/2517349.2522738

[28] Pandas. 2017. pandas Python data analysis library. http://pandas.pydata.org. (2017).

[29] X. Qian and Gio Wiederhold. 1991. Incremental Recomputation of Active Relational Expressions. *IEEE Trans. on Knowl. and Data Eng.* 3, 3 (Sept. 1991), 337–341. https://doi.org/10.1109/69.91063

[30] R [n. d.]. R project for statistical computing. http://www.r-project.org. ([n. d.]).

[31] Apache Spark. 2017. Spark Documentation. http://spark.apache.org/docs/latest. (2017).

[32] Apache Storm. 2017. Apache Storm. http://storm.apache.org. (2017).

[33] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*, Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras (Eds.). IEEE, 996–1005. http://infolab.stanford.edu/~ragho/hive-icde2010.pdf

[34] Reynold Xin et al. [n. d.]. GraySort on Apache Spark by Databricks. http://sortbenchmark.org/ApacheSpark2014.pdf. ([n. d.]).

[35] Burak Yavuz and Tyson Condie. 2017. Running Streaming Jobs Once a Day For 10x Cost Savings. https://databricks.com/blog/2017/05/22/running-streaming-jobs-day-10x-cost-savings.html. (2017).

[36] 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. 15–28.

[37] Matei Zaharia, Tathagata Das, Haoyuan Li, Tim Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *SOSP*.

[38] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. 1995. View Maintenance in a Warehousing Environment. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD '95)*. ACM, New York, NY, USA, 316–327. https://doi.org/10.1145/223784.223848