Ch. 20 - Stream Processing Fundamentals

compute this result continuously in a production setting

DStreams API

- one of the first streams APIs
- stream processing using high-level functional operators like map and reduce.
- Hundreds of organizations now use DStreams in production for large real-time applications
    - often processing terabytes of data per hour. Much like the Resilient Distributed Dataset (RDD) API, however, the
- DStreams API is based on relatively low-level operations on Java/Python objects that limit opportunities for higher-level optimization.

Structured Streaming

- only streaming covered in this book
- a new streaming API built directly on DataFrames that supports both rich optimizations and significantly simpler integration with other DataFrame and Dataset code
- Structured Streaming API was marked as stable in Apache Spark 2.2
- integrates directly with the DataFrame and Dataset APIs
- for writing new streaming applications

If you are interested in DStreams, many other books cover that API, including several dedicated books on Spark Streaming only, such as **Learning Spark Streaming** by Francois Garillot and Gerard Maas (O'Reilly, 2017).
Much as with RDDs versus DataFrames, however, Structured Streaming offers a superset of the majority of the functionality of DStreams, and will often perform better due to code generation and the Catalyst optimizer.

Stream processing

- continuously incorporating new data to compute a result
- stream processing, the input data is unbounded and has no predetermined beginning or end.
- forms a series of events that arrive at the stream processing system (e.g., credit card transactions, clicks on a website, or sensor readings from Internet of Things [IoT] devices)
- User applications can then compute various queries over this stream of events (e.g., tracking a running count of each type of event or aggregating them into hourly windows).
- The application will output multiple versions of the result as it runs, or perhaps keep it up to date in an external "sink" system such as a key-value store.
- Naturally, we can compare streaming to batch processing, in which the computation runs on a fixed-input dataset.
- Oftentimes, this might be a large-scale dataset in a data warehouse that contains all the historical events from an application (e.g., all website visits or sensor readings for the past month).
- Batch processing also takes a query to compute, similar to stream processing, but only computes the

result once.
- Although streaming and batch processing sound different, in practice, they often need to work together.
- For example, streaming applications often need to join input data against a dataset written periodically by a batch job, and the output of streaming jobs is often files or tables that are queried in batch jobs.
- application needs to work consistently across streaming and batch execution:
- Structured Streaming developers coined the term continuous applications to capture end-to-end applications that consist of streaming, batch, and interactive jobs all working on the same data to deliver an end product.
- Structured Streaming is focused on making it simple to build such applications in an end-to-end fashion instead of only handling stream-level per-record processing.

Stream Processing Use Cases

Notifications and alerting

- Given some series of events, a notification or alert should be triggered if some sort of event or series of events occurs.
- This doesn't necessarily imply autonomous or preprogrammed decision making; alerting can also be used to notify a human counterpart of some action that needs to be taken.
- An example might be driving an alert to an employee at a fulfillment center that they need to get a certain item from a location in the warehouse and ship it to a customer

Real-time reporting

- Many organizations use streaming systems to run real-time dashboards that any employee can look at.
- authors leverage Structured Streaming every day to run **real-time reporting dashboards** throughout Databricks (where both authors of this book work).
- We use these dashboards to monitor total platform usage, system load, uptime, and even usage of new features as they are rolled out, among other applications.

Incremental ETL

- reduce the latency companies must endure while retrieving information into a data warehouse—in short, "my batch job, but streaming."
- Spark batch jobs are often used for **Extract, Transform, and Load (ETL)** workloads that turn raw data into a structured format like Parquet to enable efficient queries.
- Using Structured Streaming, these jobs can incorporate new data within seconds, enabling users to query it faster downstream.
- In this use case, it is critical that data is processed exactly once and in a fault-tolerant manner: we don't want to lose any input data before it makes it to the warehouse, and we don't want to load the same data twice.
- Moreover, the streaming system needs to **make updates to the data warehouse transactionally** so as not to confuse the queries running on it with partially written data.

Update data to serve in real time

- compute data that gets served interactively by another application.
- For example, a web analytics product such as Google Analytics might **continuously track** the number of visits to each page, and use a streaming system to keep these counts up to date.
- Supporting this use case requires that the streaming system can perform incremental updates to a key–value store (or other serving system) as a sync, and often also that these updates are transactional, as in the ETL case, to avoid corrupting the data in the application.

Real-time decision making

- Real-time decision making on a streaming system involves analyzing new inputs and responding to them automatically using business logic.
- An example use case would be a bank that wants to automatically verify whether a new transaction on a customer's credit card represents fraud based on their recent history, and deny the transaction if the charge is determined fradulent.
- This decision needs to be made in real-time while processing each transaction, so developers could implement this business logic in a streaming system and run it against the stream of transactions.
- This type of application will likely need to maintain a significant amount of state about each user to track their current spending patterns, and automatically compare this state against each new transaction.

Online machine learning

- A close derivative of the real-time decision-making use case is online machine learning.
- train a model on a combination of streaming and historical data from multiple users.
- An example might be more sophisticated than the aforementioned credit card transaction use case: rather than reacting with hardcoded rules based on one customer's behavior, the company may want to continuously update a model from all customers' behavior and test each transaction against it.
- This is the most challenging use case of the bunch for stream processing systems because it requires aggregation across multiple customers, joins against static datasets, integration with machine learning libraries, and low-latency response times.

Advantages of Stream Processing

batch is much simpler to understand, troubleshoot, and write applications in for the majority of use cases.

the ability to process data in batch allows for vastly higher data processing throughput

Stream processing is essential in two cases

- stream processing enables lower latency: when your application needs to respond quickly (on a timescale of minutes, seconds, or milliseconds), you will need a streaming system that can keep state in memory to get acceptable performance. Many of the decision making and alerting use cases we described fall into this camp.
- Second, stream processing can also be more efficient in updating a result than repeated batch jobs, because it automatically incrementalizes the computation. For example, if we want to compute web

traffic statistics over the past 24 hours, a naively implemented batch job might scan all the data each time it runs, always processing 24 hours' worth of data. In contrast, a streaming system can remember state from the previous computation and only count the new data.

- If you tell the streaming system to update your report every hour, for example, it would only need to process 1 hour's worth of data each time (the new data since the last report). In a batch system, you would have to implement this kind of incremental computation by hand to get the same performance, resulting in a lot of extra work that the streaming system will automatically give you out of the box.

Challenges of Stream Processing

Imagine that our application receives input messages from a sensor (e.g., inside a car) that report its value at different times. We then want to search within this stream for certain values, or certain patterns of values. One specific challenge is that the input records might arrive to our application out-of-order: due to delays and retransmissions, for example, we might receive the following sequence of updates in order, where the time field shows the time when the value was actually measured:

```
{value: 1, time: "2017-04-07T00:00:00"}
{value: 2, time: "2017-04-07T01:00:00"}
{value: 5, time: "2017-04-07T02:00:00"}
{value: 10, time: "2017-04-07T01:30:00"}
{value: 7, time: "2017-04-07T03:00:00"}
```

In any data processing system, we can construct logic to perform some action based on receiving the single value of "5." In a streaming system, we can also respond to this individual event quickly. However, things become more complicated if you want only to trigger some action based on a specific sequence of values received, say, 2 then 10 then 5. In the case of batch processing, this is not particularly difficult because we can simply sort all the events we have by time field to see that 10 did come between 2 and 5. However, this is harder for stream processing systems.

The reason is that the streaming system is going to receive each event individually, and will need to track some state across events to remember the 2 and 5 events and realize that the 10 event was between them.

The need to remember such state over the stream creates more challenges. For instance, what if you have a massive data volume (e.g., millions of sensor streams) and the state itself is massive? What if a machine in the sytem fails, losing some state? What if the load is imbalanced and one machine is slow? And how can your application signal downstream consumers when analysis for some event is "done" (e.g., the pattern 2-10-5 did not occur)? Should it wait a fixed amount of time or remember some state indefinitely?

To summarize, the challenges we described in the previous paragraph and a couple of others, are

as follows:

- Processing out-of-order data based on application timestamps (also called event time)
- Maintaining large amounts of state
- Supporting high-data throughput
- Processing each event exactly once despite machine failures
- Handling load imbalance and stragglers
- Responding to events at low latency
- Joining with external data in other storage systems
- Determining how to update output sinks as new events arrive
- Writing data transactionally to output systems
- Updating your application's business logic at runtime

Stream Processing Design Points

Record-at-a-Time Versus Declarative APIs

- The simplest way to design a streaming API would be to just pass each event to the application and let it react using custom code.
- This is the approach that many early streaming systems, such as Apache Storm, implemented, and it has an important place when applications need full control over the processing of data.
- Streaming that provide this kind of record-at-a-time API just give the user a collection of "plumbing" to connect together into an application.
- However, the downside of these systems is that most of the complicating factors we described earlier, such as maintaining state, are solely governed by the application.
- For example, with a record-at-a-time API, you are responsible for tracking state over longer time periods, dropping it after some time to clear up space, and responding differently to duplicate events after a failure.
- As a result, many newer streaming systems provide declarative APIs, where your application specifies what to compute but not how to compute it in response to each new event and how to recover from failure.
- Spark's original DStreams API, for example, offered functional API based on operations like map, reduce and filter on streams.
- Internally, the DStream API automatically tracked how much data each operator had processed, saved any relevant state reliably, and recovered the computation from failure when needed.
- Systems such as Google Dataflow and Apache Kafka Streams provide similar, functional APIs.
- Spark's Structured Streaming actually takes this concept even further, switching from functional operations to relational (SQL-like) ones that enable even richer automatic optimization of the execution without programming effort.

Event Time Versus Processing Time

- For the systems with declarative APIs, a second concern is whether the system natively supports event time.

- Event time is the idea of processing data based on timestamps inserted into each record at the source, as opposed to the time when the record is received at the streaming application (which is called processing time).
- In particular, when using event time, records may arrive to the system out of order (e.g., if they traveled back on different network paths), and different sources may also be out of sync with each other (some records may arrive later than other records for the same event time).
- If your application collects data from remote sources that may be delayed, such as mobile phones or IoT devices, event-time processing is crucial: without it, you will miss important patterns when some data is late. In contrast, if your application only processes local events (e.g., ones generated in the same datacenter), you may not need sophisticated event-time processing
- When using event-time, several issues become common concerns across applications, including tracking state in a manner that allows the system to incorporate late events, and determining when it is safe to output a result for a given time window in event time (i.e., when the system is likely to have received all the input up to that point).
- Because of this, many declarative systems, including Structured Streaming, have "native" support for event time integrated into all their APIs, so that these concerns can be handled automatically across your whole program.

Continuous Versus Micro-Batch Execution

- The final design decision you will often see come up is about continuous versus micro-batch execution.
- In continuous processing-based systems, each node in the system is continually listening to messages from other nodes and outputting new updates to its child nodes.
- For example, suppose that your application implements a map-reduce computation over several input streams.
- In a continuous processing system, each of the nodes implementing map would read records one by one from an input source, compute its function on them, and send them to the appropriate reducer.
- The reducer would then update its state whenever it gets a new record.
- The key idea is that this happens on each individual record, as illustrated in Figure 20-1:

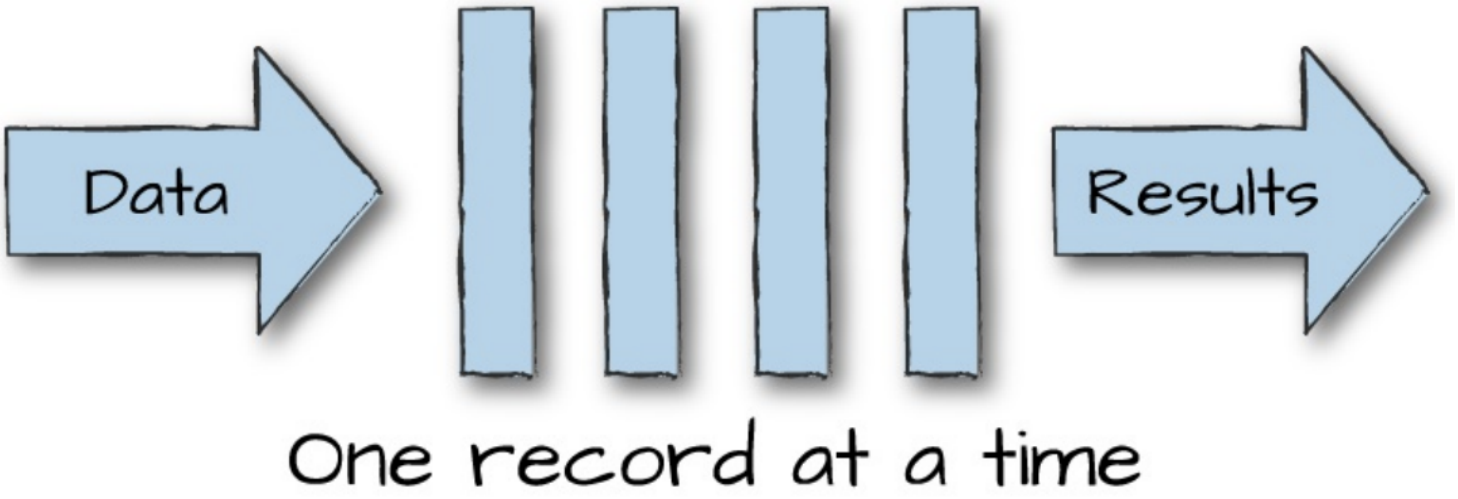

Figure 20-1. Continuous processing

- Continuous processing has the advantage of offering the lowest possible latency when the total input rate is relatively low, because each node responds immediately to a new message.
- However, continuous processing systems generally have lower maximum throughput, because they incur a significant amount of overhead per-record (e.g., calling the operating system to send a packet to a downstream node).
- In addition, continuous systems generally have a fixed topology of operators that cannot be moved at runtime without stopping the whole system, which can introduce load balancing issues.
- In contrast, micro-batch systems wait to accumulate small batches of input data (say, 500 ms' worth), then process each batch in parallel using a distributed collection of tasks, similar to the execution of a batch job in Spark.
- Micro-batch systems can often achieve high throughput per node because they leverage the same optimizations as batch systems (e.g., vectorized processing), and do not incur any extra per-record overhead, as illustrated in Figure 20-2

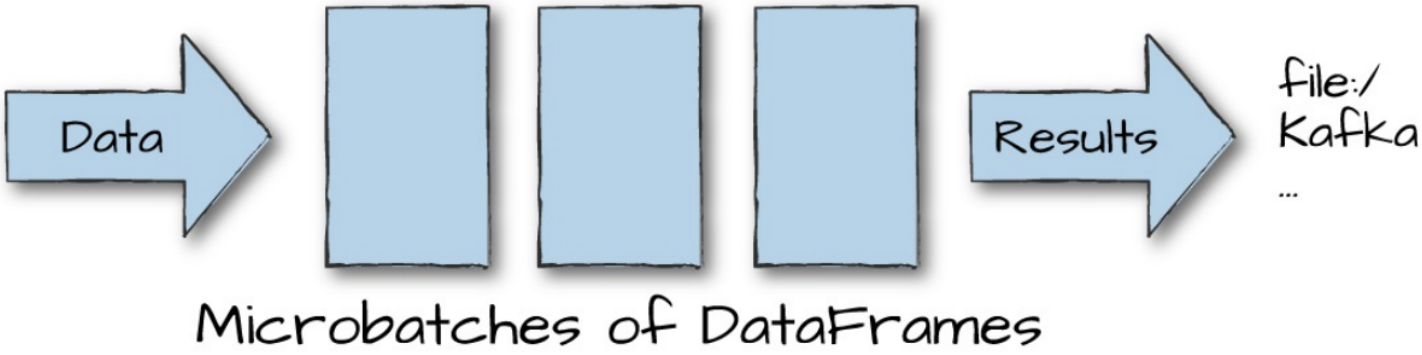

Figure 20-2. Micro-batch

- Thus, they need fewer nodes to process the same rate of data.
- Micro-batch systems can also use dynamic load balancing techniques to handle changing workloads (e.g., increasing or decreasing the number of tasks).
- The downside, however, is a higher base latency due to waiting to accumulate a micro-batch.
- In practice, the streaming applications that are large-scale enough to need to distribute their computation tend to prioritize throughput, so Spark has traditionally implemented micro-batch processing.
- In Structured Streaming, however, there is an active development effort to also support a continuous processing mode beneath the same API.
- When choosing between these two execution modes, the main factors you should keep in mind are your desired latency and total cost of operation (TCO).
- Micro-batch systems can comfortably deliver latencies from 100 ms to a second, depending on the application.
- Within this regime, they will generally require fewer nodes to achieve the same throughput, and hence lower operational cost (including lower maintenance cost due to less frequent node failures).
- For much lower latencies, you should consider a continuous processing system, or using a micro-batch system in conjunction with a fast serving layer to provide low-latency queries (e.g., outputting data into MySQL or Apache Cassandra, where it can be served to clients in milliseconds).

Spark's Streaming APIs

The DStream API

- DStream API in Spark Streaming is purely micro-batch oriented. It has a declarative (functional-based) API but no support for event time.
- Spark's original DStream API has been used broadly for stream processing since its first release in 2012.
- was the most widely used processing engine in Datanami's 2016 survey.
- Many companies use and operate Spark Streaming at scale in production today due to its high-level API interface and simple exactly-once semantics.
- Interactions with RDD code, such as joins with static data, are also natively supported in Spark Streaming. Operating Spark Streaming isn't much more difficult than operating a normal Spark cluster.
- However, the DStreams API has several limitations. First, it is based purely on Java/Python objects and functions, as opposed to the richer concept of structured tables in DataFrames and Datasets.
- This limits the engine's opportunity to perform optimizations.
- Second, the API is purely based on processing time—to handle event-time operations, applications need to implement them on their own.
- Finally, DStreams can only operate in a micro-batch fashion, and exposes the duration of micro-batches in some parts of its API, making it difficult to support alternative execution modes.

Structured Streaming

- The newer Structured Streaming API adds higher-level optimizations, event time, and support for continuous processing.
- Structured Streaming is a higher-level streaming API built from the ground up on Spark's Structured APIs.
- It is available in all the environments where structured processing runs, including Scala, Java, Python, R, and SQL.
- Like DStreams, it is a declarative API based on highlevel operations, but by building on the structured data model introduced in the previous part of the book, Structured Streaming can perform more types of optimizations automatically.
- However, unlike DStreams, Structured Streaming has native support for event time data (all of its the windowing operators automatically support it).
- As of Apache Spark 2.2, the system only runs in a micro-batch model, but the Spark team at Databricks has announced an effort called Continuous Processing to add a continuous execution mode.
- This should become an option for users in Spark 2.3.
- More fundamentally, beyond simplifying stream processing, Structured Streaming is also designed to make it easy to build end-to-end continuous applications using Apache Spark that combine streaming, batch, and interactive queries.
- For example, **Structured Streaming does not use a separate API from DataFrames: you simply write a normal DataFrame (or SQL) computation and launch it on a stream**. Structured Streaming will automatically update the result of this computation in an incremental fashion as data arrives.
- This is a major help when writing end-to-end data applications: developers **do not need to maintain a**

separate streaming version of their batch code, possibly for a different execution system, and risk having these two versions of the code fall out of sync.

- As another example, Structured Streaming can output data to standard sinks usable by Spark SQL, such as Parquet tables, making it easy to query your stream state from another Spark applications.
- In future versions of Apache Spark, we expect more and more components of the project to integrate with Structured Streaming, including online learning algorithms in MLlib.
- In general, Structured Streaming is meant to be an easier-to-use and higher-performance evolution of Spark Streaming's DStream API, so we will focus solely on this new API in this book.
- Many of the concepts, such as building a computation out of a graph of transformations, also apply to DStreams, but we leave the exposition of that to other books.