

State Management in Spark Structured Streaming



chandan prakash

Follow

9 min read · Oct 23, 2018



470



6



Stream processing means processing unbounded streams of data in real time, as and when the data arrives.

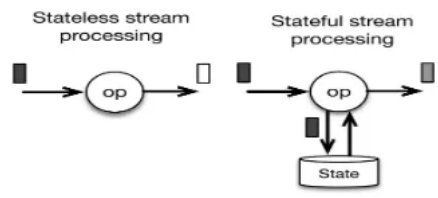
With respect to data, there are 2 ways processing can be done in streaming :

1. Stateless :

Every incoming record is independent of other records. There is no relation between different records, each record can be processed and persisted independently. e.g. Operations like map, filter, join with static data, etc come under stateless processing.

2. Stateful :

Processing of an incoming record depends upon the result of previously processed records. So we need to maintain an intermediate information between processing of different records. Every incoming record, during processing, may read and update this information. This intermediate information is called "State" in Stateful Processing. E.g. Operations like aggregating count of records per distinct key, deduplicating records, etc are examples of Stateful Processing.



State in Stream processing :

“State” is a loosely used term in stream processing world, let's understand it clearly before moving forward.

State basically means “intermediate information” that needs to be maintained for processing streams of data correctly.

Now, There are 2 types of intermediate information (“state”) in Stream Processing :

1. State of Progress (of Stream Processing) :

It is metadata of stream processing. It means keeping track of data that has been processed in streaming so far. In streaming world, we call it checkpointing/saving of offsets of incoming data. It is needed for fault tolerance in case of events like restart, upgrade, task failures. This information is bare minimum need for any reliable stream processing and is expected in both Stateless and Stateful processing.

2. State of Data (being processed in Stream Processing) :

It is the intermediate information derived from data (processed so far), that needs to be maintained between records. This is a processing need only in Stateful mode of processing.

In Streaming, when we say “state”, it usually means the intermediate data maintained between records (unless clearly mentioned about offsets or state of progress)

Need of State Store :

For maintaining State in Stateful stream processing, we need a State Store. It can be anything from a basic in-memory HashMap to persistent file systems like HDFS to distributed storage systems like Cassandra to local embedded store like RocksDb.

The purpose of a State Store is to provide a reliable place where the engine can write to and read from , the intermediary result of processing.

In this post, we will take a deep dive to understand, how State Store has been internally implemented in Structured Streaming (as of 2.3 version). Thanks to this implementation, even in the case of failure of driver or executors or both, Spark can reliably recover the stream processing state to the point

before the failure.

Although I will try to keep it as simple as I can, some basic knowledge of Spark is needed in order to understand the following details.

State Management in DStream/Old Spark Streaming :

We live in an evolving world. Something new always comes up because the older one was not good enough anymore. Lets understand what does Structured Streaming bring on table which old Spark Streaming did not. In old Spark Streaming, State Management was quite inefficient due to which it was not fit for Stateful processing. It was because of 2 major limitations in its design :

- In every micro-batch, the state was persisted along with the checkpoint metadata (i.e. offsets or progress of streaming). This was done at the end of each and every micro-batch even when there was no change in the state at all. Moreover, there was no provision of incremental persistence of state data. Every time, the snapshot of entire state was unnecessarily serialized and saved to store/file system (instead of only the part of state that changed in the micro-batch).
- Saving state to store was tightly coupled with Spark RDD tasks/jobs. It was part of Spark job to save state at the end of processing in a micro-batch. Being synchronous to RDD computation, State management caused overhead of processing delay as well as resource wastage.

Both the above limitations caused serious performance issues especially when size of state grows.

State Management in Structured Streaming :

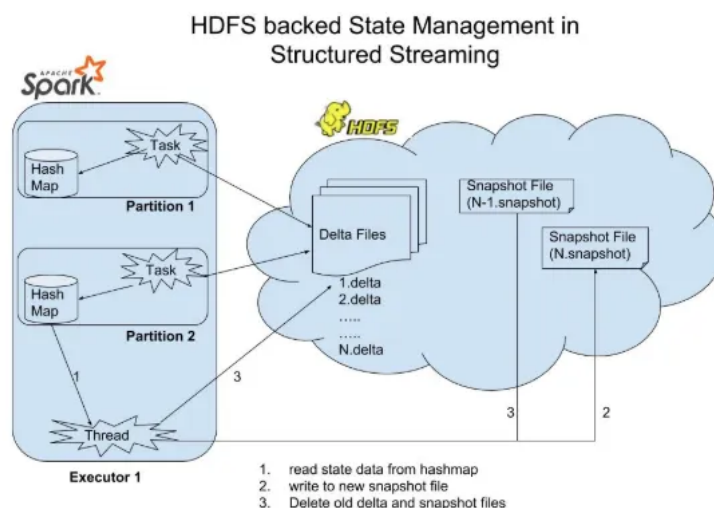
Structured Streaming, the new sql based streaming, has taken a fundamental shift in approach to manage state. It has introduced major changes to address the issues of older Spark Streaming.

Top highlight

The state management is now decoupled from metadata checkpointing and is not part of spark jobs/tasks anymore. It is asynchronous to RDD execution now and supports incremental state persistence as well.

Lets understand this in detail.

P.S. The following diagram has been drawn based on my personal understanding of Spark 2.3 code base, so please take it with pinch of salt. Feel free to comment if you have something to add/correct.



Structured Streaming, as of today, provides only one implementation of State Store: **HDFS backed State Management**

This State Store has been implemented using in-memory HashMap (in executors) and HDFS files as base for underlying storage .

The whole mechanism can be briefly explained as below :

- There is a versioned key-value store (in-memory HashMap) for every aggregated RDD partition in the associated executor memory which keeps the state data in form of key-value mappings. The store is uniquely identified with combination of: *checkpointPath + operatorId + partitionId*
checkpointPath: checkpoint location of streaming query e.g.
 /tmp/testData/checkpoint/
operatorId: every aggregator operator in streaming query like groupBy is internally assigned an unique integer value.
partitionId: id of the aggregated RDD partition generated after the aggregation.
- Version basically means the batchId. Its value is equal to the batchId.
- In every micro-batch except the very first, a partition gets a dedicated new instance of HashMap with data copied from its predecessor's HashMap (same partition from last microBatch). New updates (put/delete) are applied on top of it in the current batch/version. The updated HashMap at the end of micro-batch will serve as base in the next micro-batch and the same steps will repeat.
- Also, for a partition in a micro-batch, there is a dedicated file for recording changes made in the micro-batch in fault tolerant way. This file is called versioned delta file. It contains only the state changes in the particular batch for the associated partition only. So there are as many delta file as many partitions per batch. It is created at this unique path: *checkpointLocation/state/operatorId/partitionId/\${version}.delta*

- Task for a partition is scheduled on the executor where the HashMap for the same partition from previous microBatch is present. This is decided by the driver which keeps sufficient info about the state stores on executors.
- During a task in a micro-batch, changes for get/put/remove calls for keys are made synchronously and transactionally to the in-memory HashMap and as well as to an outputstream of versioned delta file.
- Every other operation related to state management (like snapshotting, purging, deletion, management of files, etc) is done asynchronously by a separate daemon thread on executor (called MaintenanceTask). There is one such thread per executor.
- If the task succeeds, the outputstream is closed and versioned delta file is committed to the file system like HDFS. The versioned in-memory HashMap is added to list of committed HashMaps and the version number is bumped up by 1 for the partition. The new version Id will be used by the partition in next micro-batch.
- If the task for a partition fails, the corresponding in-memory HashMap is abandoned and the delta file outputstream is cancelled. That way, no updates are recorded anywhere in memory or file system. The whole task will be freshly reattempted.
- As said, there is a separate thread (MaintenanceTask) on every executor, which runs every fixed interval (default 60 secs) and does asynchronously snapshotting of complete state of each partition from the latest versioned HashMap to disk (file name: `version.snapshot` , path: `checkpointLocation/state/operatorId/partitionId/${version}.snapshot`). So after every few batches, a snapshot file is created for each partition by this thread representing the snapshot of the complete state as of that version. This thread then deletes older delta and snapshot files prior to that version.
- Note : There cannot be multiple threads in the same executor writing to same state store or delta file. But there can be multiple executors in certain scenarios (e.g. speculative execution) having same state store loaded in memory. This means that there can be only one thread writing to an in-memory HashMap but there can be multiple threads from different executors writing to same delta file .

Pros and Cons of Current Implementation :

As we know, nothing is silver bullet. Every design comes with some pros and cons.

Pros :

- Well thought extensible abstraction and interfaces. Any new State Store implementation based on some database or external store can be written.
- Unlike earlier DStream, not in-efficient and tightly coupled with executor tasks
- Incremental checkpointing of state

Cons

- State Store, in default implementation, uses executor memory for the in-memory HashMaps. There is no division in executor memory for sharing between State Store and executor tasks. It will lead to GC and OutOfMemory issues when jobs will be running at scale depending upon shuffle tasks, size of state data and available executor memory.
- Single thread per executor responsible for snapshotting and purging of state data. With large state and too many partitions per executor, this single thread might be over-burdened with work and can lead to delay in creating snapshots and files purging.

State Management in Structured Streaming vs other Streaming Systems :

This post will be incomplete if we do not compare with the state management done in other streaming systems. Most of the other open source Streaming Systems like Flink, Samza and Kafka Streams use RocksDB to address memory limitation of state store. RocksDB addresses memory concerns but is not fault-tolerant in case of node failures. For more details on RocksDB, please refer [my last post](#).

Kafka Streams and Samza use RocksDB for unlimited fast local storage. For fault tolerance, both [Samza](#) and [KafkaStreams](#) depend on Kafka and follow a similar approach. They write the change logs for every update to some internal Kafka topic, which are log compacted time to time thus essentially becoming a single snapshot log file of entire key-value state data. In case of failures and restart, RocksDB is restored by populating from this Kafka topic.

[Flink](#) on the other hand, uses its unique snapshot strategy for fault-tolerance, instead of depending on some external system like Kafka. Time to time Flink takes snapshot of RocksDB database and copy to reliable file system like HDFS. In case of failure, RocksDB is restored from the latest snapshot. There will be some data between the time of last snapshot and the time of failure, for which state was not persisted in snapshot. In order to recover for that, the processing of the tasks in Flink operator resumes from the point of the snapshot to guarantee the unaccounted data is reprocessed.

It is important to keep in mind that this is possible only in case of replay-able data sources like Kafka, Kinesis, etc where we can go back in time to restart processing from a previous offset.

Storm/Storm Trident, as far as I know, depends on external stores like Cassandra/Redis for state management which are reliable and fault tolerant but may not be fast enough at scale. An external store comes with lot of network calls which add latency in stream processing. This is the reason why most of streaming systems use embedded local store like RocksDB.

Conclusion :

The current design of State Management in Structured Streaming is a huge forward step when compared with old DStream based Spark Streaming. It addresses the earlier issues and is a very well thought design. But there is need for a reliable state store implementation when compared with other streaming systems which can perform at scale. It will be interesting to watch how things will evolve as streaming space becomes more mature and competitive with time.

Happy Streaming!!

Follow me on [Linkedin](#) and [Quora](#)

[Big Data](#)[Streaming Analytics](#)[Spark Streaming](#)[Rocksdb](#)[Stream Processing](#)

Written by chandan prakash

484 followers · 70 following

All Things Distributed | Engine Developer | Data Engineer

Follow

Responses (6)



Aaditya Adhikari

What are your thoughts?



Mahmoud Hanafy

Feb 25, 2021



does this mean that state is stored in the execution memory? not the storage memory?



3

[Reply](#)



Ramadhana Ibnu Akbar

Jan 27, 2019



Great article! very deep understanding about spark structured streaming.



1

[Reply](#)



meow licous

Jan 17, 2019



Very insightful and great comparison to other streaming systems.

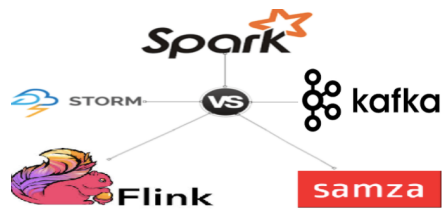



1

[Reply](#)

[See all responses](#)

More from chandan prakash




 chandan prakash

Spark Streaming vs Flink vs Storm vs Kafka Streams vs Samza :

According to a recent report by IBM Marketing cloud, "90 percent of the data in

May 1, 2018  2.1K  7  

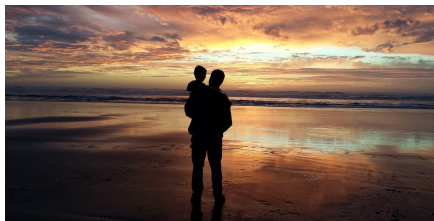


 chandan prakash

Amazon Managed Streaming for Kafka (MSK) with Spark Structured

Amazon recently announced offering Kafka Service to its AWS customers. They named it


Jan 21, 2019  38  



 chandan prakash

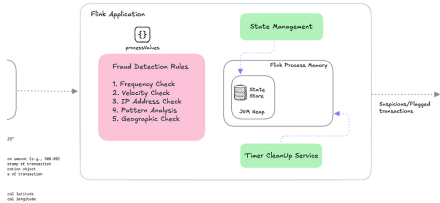
Learnings of a Father from his Kid

While writing an article, in the first few lines, I clearly mention what I am going to discuss

Dec 1, 2018  28  

See all from chandan prakash

Recommended from Medium



How Can You Optimize Spark SQL Queries?—An Interview Guide for

Whether you're getting ready for a big data engineering interview or trying to improve the

Real Time Fraud Detection Using Apache Flink—Part 1

A simple Flink application to detect suspicious transactions

Jun 14

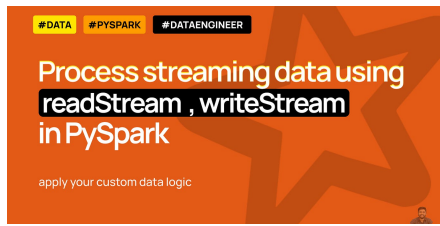
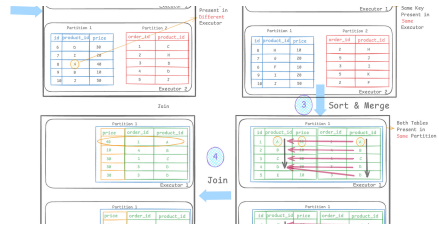
38

1

Feb 6

38

1



Sort Merge Join: Visual Flow inside Executors

Learn the default join strategy used by Spark through visuals.

Process Streaming Data in Real-Time using readStream and

Process your data when it just arrive

Dec 29, 2024

60

Feb 27



Understanding Spark's Catalyst Optimizer: Demystifying Query

Non-members can access the full article through this Link.

Mastering PySpark: Your Complete Guide to 46 Essential

Collection of PySpark Functions

Jun 17

10

Jun 3

36

See more recommendations

[Help](#) [Status](#) [About](#) [Careers](#) [Press](#) [Blog](#) [Privacy](#) [Rules](#) [Terms](#) [Text to speech](#)