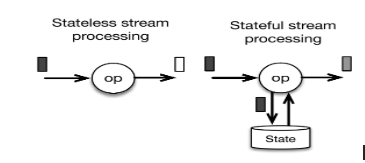
State Management in Spark Structured Streaming

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.

https://miro.medium.com/max/60/0*lA9x_FS0AxeEYZsT?q=20



# State in Stream processing :

“State” is a loosely used term in stream processing world, lets 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.

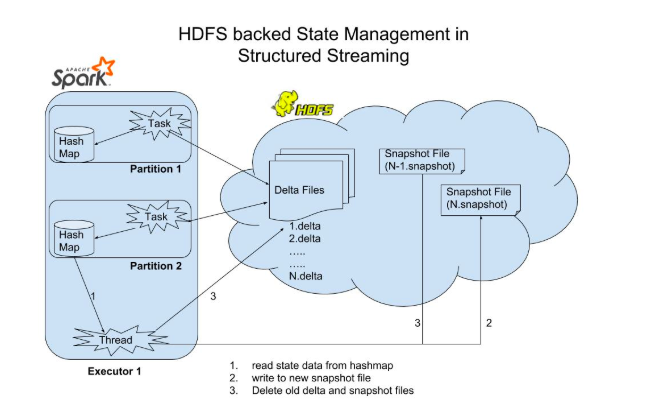
# State Management in Structured Streaming :

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.

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.

https://miro.medium.com/max/60/0*0aAdAPBJBPeZQrw-?q=20



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](https://www.linkedin.com/pulse/why-when-distributed-stream-processing-must-have-store-prakash/?source=post_page---------------------------).

Kafka Streams and Samza use RocksDB for unlimited fast local storage. For fault tolerance, both [Samza](http://samza.apache.org/learn/documentation/0.8/container/state-management.html?source=post_page---------------------------) and [KafkaStreams](https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Internal+Data+Management?source=post_page---------------------------) 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](https://ci.apache.org/projects/flink/flink-docs-stable/ops/state/state_backends.html?source=post_page---------------------------#the-rocksdbstatebackend) 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!!**

# Productionalizing Spark Streaming Applications

The [Apache Spark](http://spark.apache.org/?source=post_page---------------------------) project has become an essential tool in a Big Data Engineers toolkit. It includes many capabilities ranging from a highly performant Batch processing engine to a near-realtime Streaming Engine.

**Spark Streaming**

More and more at [Clairvoyant](http://clairvoyantsoft.com/?source=post_page---------------------------) we’ve been working with clients who are interested in building Highly Performant Real-time systems for their business. Many use cases have come up including Alert Engines, Processing IoT Data, and much more. We’ve dabbled in several types of technologies including [Apache Nifi](https://nifi.apache.org/?source=post_page---------------------------), [Apache Flume](https://flume.apache.org/?source=post_page---------------------------), [Apache Flink](https://flink.apache.org/?source=post_page---------------------------) and more. But one of our favorite technologies to use is Spark Streaming.

[Spark Streaming](https://spark.apache.org/docs/latest/streaming-programming-guide.html?source=post_page---------------------------) is an extension to Core Apache Spark that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Source data streams can be any described in the below image and more.

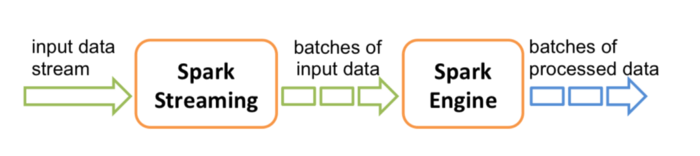
https://miro.medium.com/max/30/0*lmi5SV0Li5DhIZsN?q=20



Spark Streaming — [https://spark.apache.org/docs/latest/img/streaming-arch.png](https://spark.apache.org/docs/latest/img/streaming-arch.png?source=post_page---------------------------)

Under the covers, Spark Streaming operates with a micro-batch architecture. This means that periodically (every X number of seconds)Spark Streaming will trigger a job to be ran on the Spark Engine. During this time, Spark will pull messages from some source, process the data using the Directed Acyclic Graph (DAG) you defined, and save the data to the location you specify as a Sink.

https://miro.medium.com/max/30/0*WnpPUbI-IZET2geO?q=20



Spark Streaming Processing — [https://spark.apache.org/docs/latest/img/streaming-flow.png](https://spark.apache.org/docs/latest/img/streaming-flow.png?source=post_page---------------------------)

While implementing Spark Streaming solutions, we’ve found we’ve needed to implement some additional steps to get Spark Streaming ready for prime time on the Production cluster. This article describes the primary steps.

**Starting Point Code**

If we’re going to talk about how we can take a Spark Streaming job and get it ready for production, we first need a Spark Streaming job to improve. The bellow code will represent our starting point:

The above code does the following:

1. Creates the StreamingContext and defines the batch interval as 2 seconds
2. Establishes a Connection to Kafka and creates a DStream
3. Runs a word count on the RDDs in the DStream
4. Prints the results to the Console
5. Starts up the StreamingContext

In summary: it’s a simple word count example using Apache Kafka as its source.

**Use YARN Cluster Mode**

First lets review how you start up your spark application:

1. Build your JAR (or Python File)
2. Execute the spark-submit command:

$ spark-submit --class “org.apache.spark.testSimpleApp” --master local[4] /path/to/jar/simple-project\_2.11–1.0.jar

In the above spark-submit command, we’re specifying the **master** as local[4]. What this means is that you’re running the Spark Application in local mode and not on the cluster where the data resides.

First lets go over the Spark Architecture:

https://miro.medium.com/max/30/0*7MiRvaYNbe8M5enl?q=20

Spark Architecture — [http://blog.cloudera.com/wp-content/uploads/2014/05/spark-yarn-f1.png](http://blog.cloudera.com/wp-content/uploads/2014/05/spark-yarn-f1.png?source=post_page---------------------------)

In the above diagram, we can see that theres a Spark Driver process. This is our master process which contains all the necessary procedures and jobs that need to be executed (contains the DAGs you’ve defined in your Java, Scala, or Python code). The Driver process sends tasks to Executor processes to be completed and ensures that the tasks are completed successfully before exiting itself.

In almost every case we’ve seen, we’re usually running Spark Applications on a Hadoop Cluster where YARN (Yet Another Resource Negotiator) is available. So once you’re code is tested and ready for production, it makes sense to utilize YARN as your resource manager for allocating execution spaces for your Spark Driver and Executor processes. You can do this by specifying YARN as the master:

**Spark Version ≤1.6.3**

YARN Client Mode: *--master yarn-client*

YARN Cluster Mode: *--master yarn-cluster*

**Spark Version ≥ 2.0**

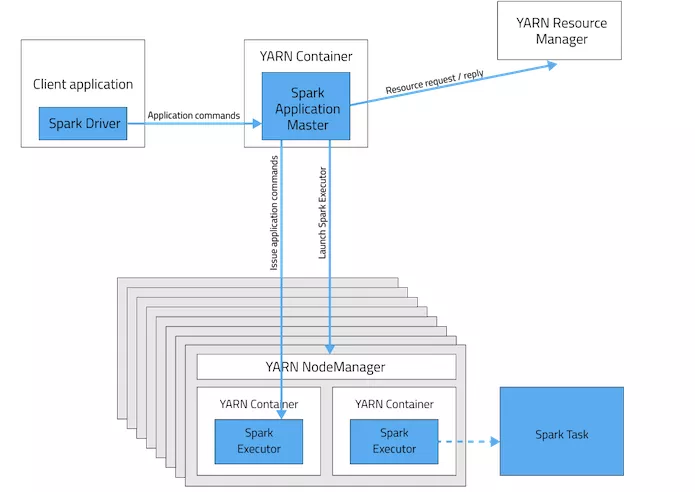
YARN Client Mode: *--master yarn --deploy-mode client*

YARN Cluster Mode: *--master yarn --deploy-mode cluster*

Above I listed 2 modes: Client and Cluster. The difference between the 2 is basically: where is the Spark Driver Running. On the Client or the Cluster. Lets go into more detail:

**YARN Client Mode**

https://miro.medium.com/max/30/0*H0TfR7Cjywx1ZDCm?q=20



YARN Client Mode — [https://4.bp.blogspot.com/-lFcEx4GDEg4/WMgZQRjDRrI/AAAAAAAADt0/SA1v6gtRGGknkmTINUWbCg5ufEM7rVb9gCLcB/s1600/SparkYanrClusterMode.jpg](https://4.bp.blogspot.com/-lFcEx4GDEg4/WMgZQRjDRrI/AAAAAAAADt0/SA1v6gtRGGknkmTINUWbCg5ufEM7rVb9gCLcB/s1600/SparkYanrClusterMode.jpg?source=post_page---------------------------)

In Client mode, the Spark Driver is running on the Client machine (or the same machine you submitting the spark-submit command from). We see most organizations running all of their Spark Applications in this mode. In the case where you’re running Spark Batch processes, this is an Ok practice. However, there’s a problem with doing the same with Spark Streaming Applications.

Spark Streaming Applications are processes that will be essentially running forever. But what if the machine where the Spark Streaming Application is running goes down? This would result in the Application itself being killed off.

**YARN Cluster Mode**

https://miro.medium.com/max/30/0*ZqFJtv5IxJE81_Li?q=20

YARN Cluster Mode — [https://4.bp.blogspot.com/-lFcEx4GDEg4/WMgZQRjDRrI/AAAAAAAADt0/SA1v6gtRGGknkmTINUWbCg5ufEM7rVb9gCLcB/s1600/SparkYanrClusterMode.jpg](https://4.bp.blogspot.com/-lFcEx4GDEg4/WMgZQRjDRrI/AAAAAAAADt0/SA1v6gtRGGknkmTINUWbCg5ufEM7rVb9gCLcB/s1600/SparkYanrClusterMode.jpg?source=post_page---------------------------)

In Cluster mode, the Spark Driver is running within a container in YARN. So you can trust YARN to handle any failures. If the machine where the Driver is running goes down, then it will be automatically restarted on another node.

**Useful Configurations**

*spark.yarn.maxAppAttempts*

The maximum number of attempts that will be made to submit the application. It should be no larger than the global number of max attempts in the YARN configuration.

*spark.yarn.am.attemptFailuresValidityInterval*

Defines the validity interval for AM failure tracking. If the AM has been running for at least the defined interval, the AM failure count will be reset. This feature is not enabled if not configured.

So if we were to set the above configurations to the values:

* spark.yarn.maxAppAttempts=2
* spark.yarn.am.attemptFailuresValidityInterval=1h

This would mean that every 1 hour it will attempt to start the App 2 times.

**Setting the Configurations**

From the spark-submit Command:

$ spark-submit--class "org.apache.testSimpleApp"--master yarn--deploy-mode cluster**--conf spark.yarn.maxAppAttempts=2--conf spark.yarn.am.attemptFailuresValidityInterval=1h**/path/to/jar/simple-project\_2.11-1.0.jar

In Code:

**Gracefully Shutdown your Streaming Application**

Now that we’re starting up the Application in the correct mode, we should discuss how to Shutdown the Spark Streaming Application in the event we want to role new features, make configuration changes, etc.

Currently through YARN you can shutdown or kill a Spark Streaming Application with the following command:

$ yarn application -kill {ApplicationID}

However, what if a Spark Streaming micro-batch is processing when you run this command and kill the application?

The short answer is: the data you’re processing would be lost.

In addition, with how Spark pulls messages from Kafka (acknowledge to Kafka that it has received the messages before processing them), when you restart your Spark Streaming Application it will skip those messages that were being processed and start processing the message that came in after.

To solve this we need to implement a Graceful Shutdown process to ensure that the Spark Streaming Application will only shutdown between micro-batch’s so we don’t loose any data.

Our first step to accomplish graceful shutdown with our starting point code above is:

// Start the computationssc.start()*ssc.awaitTermination()* **<--- REMOVE THIS LINE**

To Gracefully Shutdown the Spark Streaming Application we’ll instead follow these steps:

1. On Spark Streaming Startup: Create a touch file in HDFS
2. Within the Spark Code: Periodically check if the touch file still exists. If the touch file doesn’t exist, start the Graceful Shutdown process.
3. To Stop: Delete the touch file and wait for the Graceful Shutdown process to complete

Tip: Build a shell script to do these start and stop operations

The Spark code that you would write would look something like this:

First change is to add a global variable which indicates if we’re starting to shut the application down. Then replace the awaitTermination process with the while loop. Within the Loop we’ll periodically check if the file in HDFS exists. If it does not, then we’ll set the global variable to true and the logic within the while loops executes the stop command in the StreamingContext.

**Monitor your Streaming Application**

Like with any important application, you want to make sure that it’s running and that it’s running correctly. There are a few options for doing this with your Spark Streaming Application:

**Operational Monitoring**

See this page for more information: [http://spark.apache.org/docs/latest/monitoring#metrics](http://spark.apache.org/docs/latest/monitoring?source=post_page---------------------------#metrics)

**StreamingListener (Spark ≥ 2.1)**

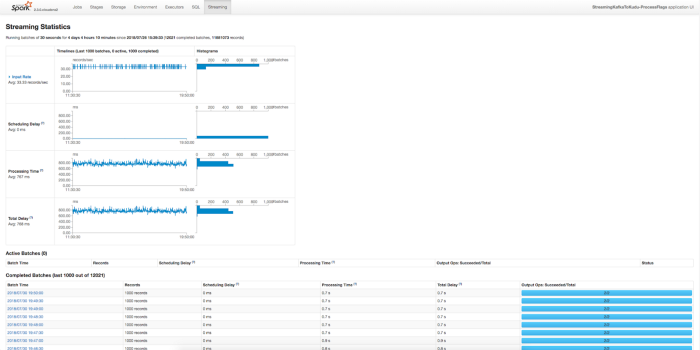
Apache Spark 2.1 and greater has added functionality to provide Listeners that trigger events at various phases of the startup and execution of the Spark Streaming of Application. Here are some of the Listeners:

* onBatchSubmitted
* onBatchStarted
* onBatchCompleted
* onReceiverStarted
* onReceiverStopped
* onReceiverError

With the above Listeners, you can manually implement a process to push various metrics to whatever monitoring service your organization uses. We’ve used it in the past to push metrics about each micro-batch (Number of Messages Received, Time to Process, Various Errors, etc.) into a relational database that we later query to ensure the process is running and its running with acceptable performance.

**Spark UI**

https://miro.medium.com/max/30/0*QBCZt04QBaM9R1SK?q=20



Spark UI

The Spark UI comes out of the box with Apache Spark and contains some very useful information. Above is just one of the views, and perhaps the most important high level view for Streaming. It provides information about each micro-batch. This includes how many records the micro-batch processed, how long it took, how much delay there was in starting up the micro-batch and more. In general this is a great way to confirm that your Spark Streaming Application is running and that its running with the correct performance.

**Use Checkpointing**

You may already be familiar with the standard form of Checkpointing used in Apache Sparks Batch offering. Where the data contained within an RDD or DataFrame is persisted to disk between the execution of the tasks contained within the RDD. This ensures that if there are any executor failures, then Spark can just restart from this Checkpoint rather than restarting the execution of the RDD or DataFrame from the beginning.

While this feature is still usable in Spark Streaming, there is another form of Checkpointing that is available for Spark Streaming Applications that may be useful:

**Metadata Checkpointing**

This involves saving the Metadata defining the streaming computation to a fault-tolerant storage like HDFS. It is used to recover from the failure of the node running the driver of the Spark Streaming Application. Some of this Metadata includes:

* Configurations — The configuration that was used to create the streaming application.
* DStream operations — The set of DStream operations that define the streaming application.
* Incomplete batches — Batches whose jobs are queued but have not completed yet.

This form of Checkpointing is actually also required in the event you want to perform any Stateful Transformations like updateStateByKey or reduceByKeyAndWindow.

How you can Enable Checkpointing in your code is to do the following:

**Problems with Metadata Checkpointing**

Some things to watch out for when using Metadata Checkpointing are:

*Checkpoints can’t survive Spark Version Upgrades*

In the event you upgrade your Spark Version, you will need to manually delete your checkpoint.

*Checkpoints need to be cleared between Code Upgrades*

Since Metadata Checkpointing involves checkpointing the actual DStream operations that need to be performed on the records that are coming in, you will need to clear the checkpoint so the new operations from the Code Upgrade can be loaded. If you were to adjust your code to use slightly different operations and redeploy but fail to clear your checkpoint, the old operations would be loaded from the Checkpoint and your new changes would not be reflected.

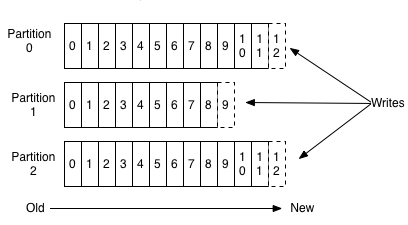
**Setup Multiple Partitions in your Kafka Topics**

Assuming you’re using Kafka as source of data for the Spark Streaming Application, it would really help with performance to define multiple partitions when creating your Kafka topic. Let me explain how through the following diagrams.

When you’re running the command to create the Kafka topic, you can define the number of partitions with the highlighted option in the bellow command:

kafka-topics --zookeeper <host>:2181 --create --topic <topic-name> -**-partitions <number-of-partitions>** --replication-factor <number-of-replicas>

https://miro.medium.com/max/30/0*_yrTt61NkoeLjTxs?q=20



Kafka Writes — [https://www.analyticshut.com/wp-content/uploads/2018/04/topic.png](https://www.analyticshut.com/wp-content/uploads/2018/04/topic.png?source=post_page---------------------------)

When data is pushed into a Kafka topic, the data is automatically distributed across the partitions by the Key you define in the Kafka Message. Each message is added to the Kafka topic with an offset associated with it, or an ID that indicates its position in the partition. If you were to specify null as the Key, the message will be automatically distributed evenly across the partitions.

https://miro.medium.com/max/30/0*npF0aBHGQ400X3s-?q=20

Kafka Reads — [https://fizalihsan.github.io/technology/kafka-partition-consumer.png](https://fizalihsan.github.io/technology/kafka-partition-consumer.png?source=post_page---------------------------)

The above diagram shows how the Spark Streaming Application can work when it’s processing messages from a Kafka topic with multiple partitions. Each “Consumer” can be thought of as one of the Spark Executors. Each Spark Executor can independently load data from a specific Kafka topic, rather than a single source. In addition, each partition can also exist on a different Kafka Broker instance (separate node), which will help to decrease the load on any one node.

**Use Direct Streams with Kafka**

If you’re using Kafka, then whats useful to know is that there are actually 2 types of connectors: a Receiver based Stream and a Direct Stream. I’m sure if the title of this section didn’t give it away, you might ask “Which one should I use?” and also possibly “Whats the difference?”. We’ll go over the difference here.

**Receiver Based Streaming**

https://miro.medium.com/max/30/0*o3kozg5eAjFxxJHA?q=20

Spark Streaming Receiver Based Streaming — [https://databricks.com/wp-content/uploads/2015/03/Screen-Shot-2015-03-29-at-10.11.42-PM.png](https://databricks.com/wp-content/uploads/2015/03/Screen-Shot-2015-03-29-at-10.11.42-PM.png?source=post_page---------------------------)

Receiver based streaming is actually Spark Streamings standard implementation of how it receives data from any source (similar implementations for sources like Twitter, Kinesis, etc). Each Executor has a Receiver instance running inside of it. On the start of a micro-batch, the Driver launches a job to the Executor. The Receiver process within each Executor is then triggered, which uses the Kafka High Level API to load in the latest data from the Kafka Topic. The data from the Receiver is then stored in a Write Ahead Log (WAL) before Kafka is updated the data has been received (this prevents against a loss of data). Once the data is safely in the WAL, the Spark Executors then go to work processing the messages.

How this strategy then prevents against failure is that if one of the Executors fails, then a new Executor is spawned to take its place. The Executor then loads the data the failed Executor loaded into the WAL.

Note: if you want to move forward using the Receiver based stream, you should do the following:

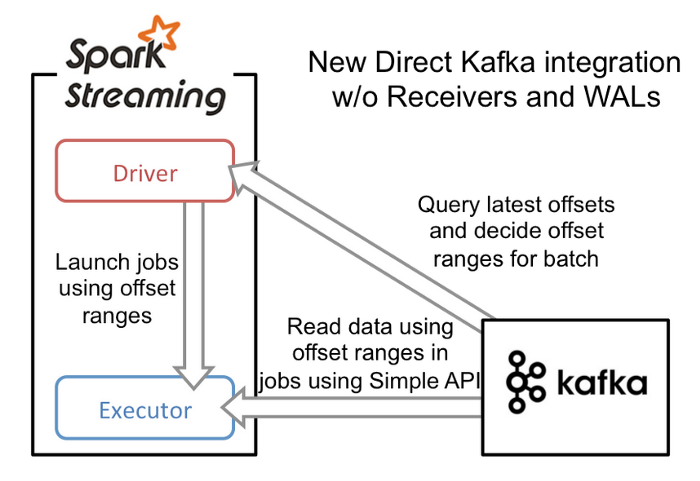
* **Enable Checkpointing** — This will allow Write Ahead Logs to be written to the Checkpoint Directory
* **Enable the WAL**— By default the WAL is not enabled in Receiver based streaming. You can enable it by specifying the configuration: *spark.streaming.receiver.wrteAheadLog.enable=true*
* **Set the proper StorageLevel for the WAL** — Since data is already persisted to HDFS, you can disable in-memory replication to avoid duplicate storage: *StorageLevel.MEMROY\_AND\_DISK\_SER*

This implementation was the first available in Spark Streaming and of course continues to work. So, why do we need another implementation?

Well here’s a thought: Kafka already stored replicated copies of the data in a circular buffer and is Highly Available. Why then do we need a WAL? Including a WAL actually decreases performance slightly since it needs to write the data to disk after receiving it from Kafka. Hence why the following option was introduced:

**Direct Stream**

https://miro.medium.com/max/30/0*OduDeFmfh7Pm_vuk?q=20



Spark Streaming Direct Stream — [https://databricks.com/wp-content/uploads/2015/03/Screen-Shot-2015-03-29-at-10.14.11-PM.png](https://databricks.com/wp-content/uploads/2015/03/Screen-Shot-2015-03-29-at-10.14.11-PM.png?source=post_page---------------------------)

In a direct stream, we remove the WAL in favor of letting Kafka act as the WAL. The execution starts with the Driver launching jobs to the executors. The Driver also passes a range of offsets to each executor that it needs to process. For example: Executor 1 might be passed the offset range *2000–2050* and Executor 2 might be passed the offset range *2051–2100*. Each Executor will load in its designated range of data and process it.

How this strategy then prevents against failure is that if one of the Executors fails, then a new Executor is spawned to take its place. The Driver will pass the exact same range to the new Executor as the previous one and retry processing that data.

**Save your Kafka Offsets**

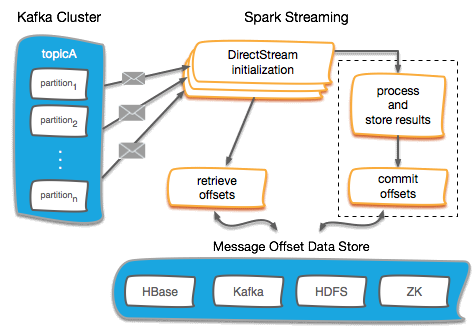
Most organizations and subgroups within said organizations we encounter have their hearts set on obtaining an **Exactly-Once Delivery Semantic** with Spark Streaming. The unfortunate thing is that it’s actually very difficult to do in a distributed fault tolerant system. It certainly doesn’t come out of the box with Spark Streaming (what you get is closer to either **At-Least-Once** or **At-Most-Once** Semantics). However, it is possible to do with Spark Streaming if you make some changes.

The main thing you need to do is Save the Kafka Offsets that were successfully processed after a micro-batch and load the last completed Kafka Offsets when you start up your Spark Streaming Application. The save operation would need to be done if and only if the transaction on the incoming messages completed successfully. To be more specific, you should store offsets after an idempotent output OR store offsets in an atomic transaction alongside output.

What this all results in is that only after the data has been transformed and saved to your output source, will you then move on from that data set. With the changes mentinoed above using Direct Streams you should then be able to process all the data in a micro-batch in a fault tolerant way and achieve the desired **Exactly-Once Delivery Semantic**.

The below diagram shows how your application can work to accomplish this.

https://miro.medium.com/max/30/0*v5spMOfLsDHRIShi?q=20



Managing Kafka Offsets — [http://blog.cloudera.com/wp-content/uploads/2017/06/Spark-Streaming-flow-for-offsets.png](http://blog.cloudera.com/wp-content/uploads/2017/06/Spark-Streaming-flow-for-offsets.png?source=post_page---------------------------)

Below is a code example that shows how you can initialize your Kafka DStream by loading in offsets from the loadOffsets Command:

It’†s assuming you’re using Kudu to store the offsets (hence the *kuduContext*), but the overall procedure can work for any storage system for your offsets: Zookeeper, HBase, HDFS, Hive, Impala, etc.

**Stabilize your Streaming Application**

Before releasing your application into production, it’s a very good idea to take some time to do some performance testing. The main thing that you will need to ensure is:

*Your Average Batch Processing Time should be less then your Batch Interval*

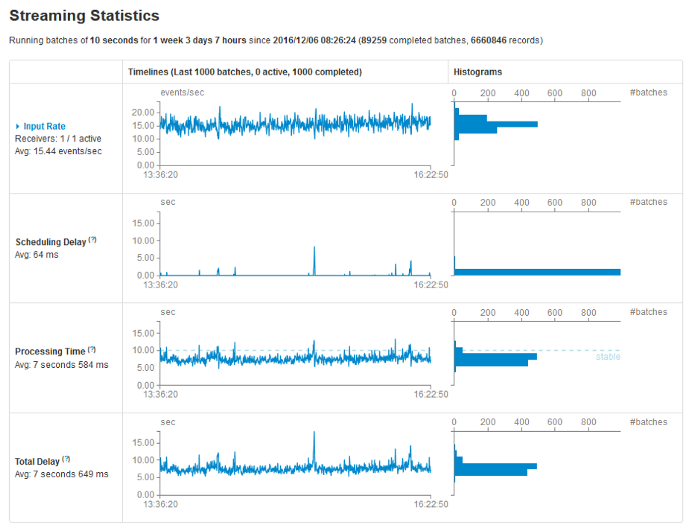
So for instance if you set your Batch Interval to 30 seconds, then the average processing time for each micro-batch should be below 30 seconds.

Only one micro-batch is executed at any given point in time. So if your first micro-batch takes 40 seconds to run, then there will be a delay before the second micro-batch actually triggered. If your micro-batches are consistently 40 seconds or so, then micro-batches will be scheduled faster than they will be completed. Eventually after a couple hours you’ll have dozens of micro-batches that are waiting to trigger.

The danger with this is that micro-batches will continue to be scheduled which will fill up the Heap. Eventually the Spark Streaming Application will fail.

You can see how much time your micro-batches are taking on the Spark UI. The bellow images show a job that has its Batch Interval set to 10 seconds and the processing time occasionally spiking above 10 seconds. Which in turn shows a spike in Scheduling Delay.

https://miro.medium.com/max/30/0*zE-5D45Z7yImkO7f?q=20



If you do encounter a problem where your micro-batches are consistently longer then your Batch Interval, are a few strategies you can employ:

**Optimize your Operations (Transformations, Joins and Writes)**

It would really be worth it to review what operations you’re doing in your Spark Streaming Application. If you’re saving data to some external database that requires indexing, the save operation might be dramatically impacting your performance. In addition, you can also check how efficient your joins are and see if you can optimize which data set is on the right and left.

**Implement Caching**

If you’re processing the same source RDD/DataFrame multiple times, then a big improvement might be to Cache the result of the RDD/DataFrame in memory.

**Increase parallelism**

You could be running into a bottleneck situation with Kafka where you don’t have enough partitions. If you only have one partition defined in your topic or too few brokers, then this could be reducing the time to deliver the data to the Spark Executors.

It could also be that your Spark Streaming Application doesn’t have enough Executors to properly process all the data thats coming in as efficiently as possible. Adding more Executors may help.

**Repartition you Data**

If you find that your data is unbalanced (all the data seems to be going to a single executor) then you should go back to optimizing your operations and see if theres a more efficient join or other operation you can do to ensure it’s balanced. Worst case scenario though, you can Repartition your data to ensure it gets redistributed:

dstream.repartition(100)

**Increase Batch Duration**

If all else fails, you could try increasing your Batch Duration. Grated this will increase the amount of data you’re processing. But, you may be performing some operation that always requires 10 seconds or so to complete, no matter how much data you’re processing. If thats the case then increasing your Batch Duration might help.

**Happy Streaming!!!**