AdaltasAdaltas

Spark Streaming part 2: run Spark Structured Streaming pipelines in Hadoop

By Oskar RYNKIEWICZ
May 28, 2019

Categories: Data Engineering, Learning | Tags: Spark, Apache Spark Streaming, Streaming, Python [more]

Spark can process streaming data on a multi-node Hadoop cluster relying on HDFS for the storage and YARN for the scheduling of jobs. Thus, Spark Structured Streaming integrates well with Big Data infrastructures. A streaming data processing chain in a distributed environment will be presented. Cluster environment demands attention to aspects such as monitoring, stability, and security.

This is the second part of a four-part series:

- In the first part, a data pipeline with a Spark Structured Streaming application is built.
- The second part concerns the migration of the pipeline to a Hadoop cluster.
- In the third part, the PySpark application was ported to Scala Spark and unit tested.
- The fourth and last part enriches the data pipeline with a Machine Learning clustering algorithm.

The same Taxi data from Apache Flink Training is used as in the first part. The use case also remains identical - the identification of the Manhattan neighborhoods that are most likely to yield high tips. The complete code from the last part could be downloaded from here.

Specification of the Hadoop cluster

Spark is well adapted to use Hadoop YARN as a job scheduler. Hadoop clusters are common execution environment for Spark in companies using Big Data technologies based on a Hadoop infrastructure. In this article, Spark on YARN is used on a small cluster with the below characteristics.

- Hadoop cluster with HDP 3.1.0 installed through Apache Ambari 2.7.3
- Hadoop deployed locally on a 4 VMs cluster running on a laptop with 32GB and 8 cores
- The VMs have respectively 5/6/7/7 GB RAM, 1/1/3/2 cores, and all are running CentOS
- Host `master01.cluster` runs Ambari server, ZooKeeper, YARN, HDFS, Hive, Tez
- Host `master02.cluster` runs Spark2 (Spark2 History Server) and Kafka (Kafka Broker)
- Hosts `worker01.cluster` and worker02.cluster are computing nodes
- SSH keys for root are set up, i.e. root user of each host has the appropriate key in `/root/.ssh`
- Each host has Python 3 installed concurrently to Python 2, with Python 3 specified to use for PySpark (set with `PYSPARK_PYTHON` Spark environment variable)
- YARN containers: the memory allocated for containers is 6144MB, with a container memory threshold from the minimum value of 512MB to the maximum value of 6144MB
- ZooKeeper server is listening on `master02.cluster:2181` and Kafka Broker on `master02.cluster:6667`

- Spark's Driver program output tends to be rich in `INFO` level logs, which adds clutter to console output. In "Advanced spark2-log4j-properties" `INFO` could be changed to `WARN` in "log4j.rootCategory" to make Spark output less verbose. When things don't work it's better to reverse to `INFO` log level

Notice that:

- At least HDP 2.6.5 or CDH 6.1.0 is needed, as stream-stream joins are supported from Spark 2.3. Actually, Spark Structured Streaming is supported since Spark 2.2 but the newer versions of Spark provide the stream-stream join feature used in the article
- Kafka 0.10.0 or higher is needed for the integration of Kafka with Spark Structured Streaming
- Defaults on HDP 3.1.0 are Spark 2.3.x and Kafka 2.x

A cluster complying with the above specifications was deployed on VMs managed with Vagrant. Each OS had environment prepared for Ambari with *Vagrantfile* and shell bootstrap files are provisioned. The installation of the components was carried out manually with Ambari itself. We also encourage you to use the Open Source project Jumbo developed by Adaltas which does just that, easily provisioning a distributed and secure Hadoop cluster. But at the time of this writing, it allows deploying local Hadoop clusters with HDP 2.6.4, which doesn't have the minimal version of Spark available.

Exposing streaming results to JDBC clients

In part 1 of this article, streaming results were printed to the console. Only the developer was being able to see them. In a production context, an application needs an interface to expose the results to the users. A possible solution to make the results available with low latencies is mounting them in the application's memory and exposing through JDBC/ODBC interface. This provides real-time access to the data but it does not persist them. In addition to exposing the data kept in-memory, persistent data storage on disk is needed which is covered later in the article.

Streaming results stored in Spark Driver's memory could be made accessible by JDBC/ODBC with Spark Thrift Server. Most of the Business Intelligence tools support JDBC/ODBC and would be able to receive streaming results. Spark applications are resource independent, hence standalone Spark Thrift Server wouldn't be able to expose data cached in different Spark application. However, it's possible to embed Spark Thrift Server in the same Spark application that runs the streaming query. The complete code from the last part shall be modified slightly.

Firstly, Spark Thrift Server should be started before the execution of the query:

```
from py4j.java_gateway import java_import
java_import(spark.sparkContext._gateway.jvm, "")
spark.sparkContext._gateway.jvm.org.apache.spark.sql.hive.thriftserve
  .HiveThriftServer2.startWithContext(spark._jwrapped)
```

Secondly, `tips.writeStream.format("console")` needs to be modified to `tips.writeStream.format("memory")`:

```
tips.writeStream \
  .outputMode("append") \
  .format("memory") \
  .queryName("tips") \
  .option("truncate", False) \
  .start() \
  .awaitTermination()
```

The main limitation of this method is the amount of RAM available for the Spark Driver. A memory sink works only for tables that could fit in the Driver's memory. Spark Driver could have only up to 200 GB because JVM wouldn't handle more. Out of these 200 GB, only a small fraction could be used for caching tables. With default values of `spark.memory.fraction` and `spark.memory.storageFraction`, the maximal amount of memory available for storage is 200GB * 0.6 * 0.5 = 60GB.

A distributed in-memory data storage would be needed to provide low latency queries on bigger datasets. Storage on a single JVM wouldn't be sufficient. Examples of data stores allowing in-memory storage and high-performance are Druid, Apache Kudu, and Apache Ignite.

Submitting the application

Spark streaming application could be submitted to be launched on the cluster with a command:

```
# as spark user from master02.cluster host
spark-submit \
--master yarn --deploy-mode client \
--num-executors 2 --executor-cores 1 \
--executor-memory 5g --driver-memory 4g \
--packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.3.0 \
--conf spark.sql.hive.thriftServer.singleSession=true \
/vagrant/sstreaming-spark-app.py
```

The `spark-submit` script takes a number of parameters specifying how the application will run on the cluster:

- Spark's executors and driver parameters are adjusted to the scarce cluster's resources presented above. Parameters are picked with precision since misconfiguration could either starve or kill the application. Only 2 executors with 5GB and 1 core each are feasible. Some considerations in choosing those values:
  - Application Master (AM) is a process enabling communication between Spark Application and Yarn's Resource Manager. The AM needs a container with at least 1024MB RAM and 1 core
  - Each node has 1GB RAM and 1 core reserved for the functioning of the OS
  - Each JVM (executors, the driver, the AM) has overhead of at least 384MB
  - Allocations are rounded to the minimal container size of 512MB
  - To account for high memory needs of stream processing and in-memory stream sink, driver's memory is set to 4GB instead of the default 1GB
  - The suggested parameters in this article assume only one such application on the cluster
  - If anything else is running on a cluster (e.g. Tez processing job), the submitted Spark app will have to wait for resources
- The `--deploy-mode` option specifhrift server runs in multi-sessioprogram runs. Submitting the application in the "client" mode places the Driver on the client submitting the Application, `master02.cluster` node. It allows observing Driver's stdout easily from the console on `master02` hosts. The "cluster" deploy mode would put the Driver in the Application Master on one of the workers in the cluster. There is no interest in doing so since `master02.cluster` is already in the cluster and there are no network latencies to combat
- The *spark-sql-kafka* package is mandatory for Spark integration with Kafka. The version at the very end has to match the version of Spark being used

- By default, the Thrift server runs in multi-session mode. Setting `spark.sql.hive.thriftServer.singleSession=true`, ensures that JDBC/ODBC client will connect to the existing session and will be able to access the temporary tables. Persisted tables are always accessible even in multiple sessions.

Previewing the results

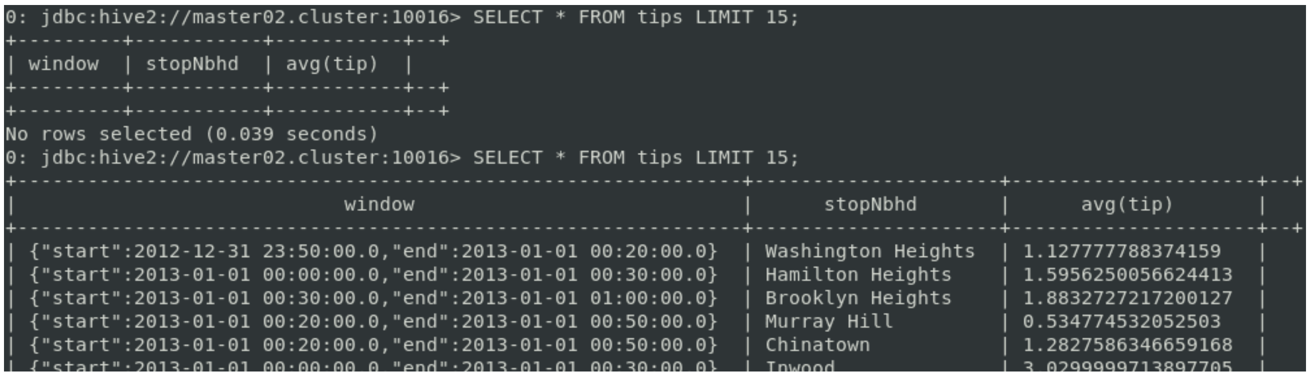
Once the application is up, the streams of data could be started. The commands from part 1 of this article are:

```
# as kafka user from master02.cluster host
( curl -s https://training.ververica.com/trainingData/nycTaxiRides.gz
split -l 10000 --filter="/usr/hdp/current/kafka-broker/bin/kafka-cons
( curl -s https://training.ververica.com/trainingData/nycTaxiFares.gz
split -l 10000 --filter="/usr/hdp/current/kafka-broker/bin/kafka-cons
```

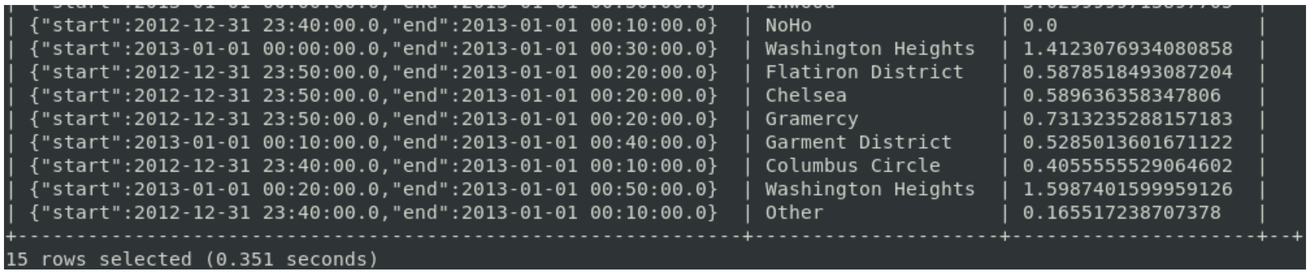
Any JDBC client should now be able to access the results. For instance, a Hive's beeline shell tool could be used to connect with Spark Thrift Server. Assuming that Spark has `hive.server2.thrift.port` property set to port 10016, beeline could connect with Spark Thrift Server with the command:

```
/usr/hdp/current/spark2-thriftserver/bin/beeline \
  -u jdbc:hive2://master02.cluster:10016
```

Notice that the beeline command provided by the Spark Thrift Server is used, not the default Hive's beeline command. Spark is compiled with support for Hive 1.2.1, while HDP 3.1 has Hive 3.1.0 installed. Using beeline from a more recent version of Hive would lead to errors due to the compatibility mismatches. Hence, an older beeline client from Spark Thrift Server has to be used. Streaming results should now be available as a temporary table. It could be queried to get the current state of the results.



```
0: jdbc:hive2://master02.cluster:10016> SELECT * FROM tips LIMIT 15;
+---------+-----------+-----------+--+
| window  | stopNbhd  | avg(tip)  |
+---------+-----------+-----------+--+
+---------+-----------+-----------+--+
No rows selected (0.039 seconds)
0: jdbc:hive2://master02.cluster:10016> SELECT * FROM tips LIMIT 15;
+---------------------------------------------------------------+----------------------+----------------------+--+
|                            window                             |       stopNbhd       |       avg(tip)       |
+---------------------------------------------------------------+----------------------+----------------------+--+
| {"start":2012-12-31 23:50:00.0,"end":2013-01-01 00:20:00.0}  | Washington Heights  | 1.127777788374159    |
| {"start":2013-01-01 00:00:00.0,"end":2013-01-01 00:30:00.0}  | Hamilton Heights    | 1.5956250056624413   |
| {"start":2013-01-01 00:30:00.0,"end":2013-01-01 01:00:00.0}  | Brooklyn Heights    | 1.8832727217200127   |
| {"start":2013-01-01 00:20:00.0,"end":2013-01-01 00:50:00.0}  | Murray Hill         | 0.534774532052503    |
| {"start":2013-01-01 00:20:00.0,"end":2013-01-01 00:50:00.0}  | Chinatown           | 1.2827586346659168   |
| {"start":2013-01-01 00:00:00.0,"end":2013-01-01 00:30:00.0}  | Inwood              | 3.029999713897705    |
```



```
| {"start":2012-12-31 23:40:00.0,"end":2013-01-01 00:10:00.0}  | NoHo                 | 0.0                  |
| {"start":2013-01-01 00:00:00.0,"end":2013-01-01 00:30:00.0}  | Washington Heights   | 1.4123076934080858   |
| {"start":2012-12-31 23:50:00.0,"end":2013-01-01 00:20:00.0}  | Flatiron District    | 0.5878518493087204   |
| {"start":2012-12-31 23:50:00.0,"end":2013-01-01 00:20:00.0}  | Chelsea              | 0.589636358347806    |
| {"start":2012-12-31 23:50:00.0,"end":2013-01-01 00:20:00.0}  | Gramercy             | 0.7313235288157183   |
| {"start":2013-01-01 00:10:00.0,"end":2013-01-01 00:40:00.0}  | Garment District     | 0.5285013601671122   |
| {"start":2012-12-31 23:40:00.0,"end":2013-01-01 00:10:00.0}  | Columbus Circle      | 0.4055555529064602   |
| {"start":2013-01-01 00:20:00.0,"end":2013-01-01 00:50:00.0}  | Washington Heights   | 1.5987401599959126   |
| {"start":2012-12-31 23:40:00.0,"end":2013-01-01 00:10:00.0}  | Other                | 0.165517238707378    |
+--------------------------------------------------------------+----------------------+----------------------+--+
15 rows selected (0.351 seconds)
```

Persisting the data in HDFS

The memory storage presented above is volatile and once the events have been processed, there is no way to restore them once the Spark application is stopped. In order to persist the incoming events, data must be persisted. The raw dataset must be stored in a storage area. It is possible to persist the transformed dataset as well or you could rebuild it from the raw dataset. There are multiple destinations, such as RDBMS databases, NoSQL stores, object cloud storage (such as Amazon S3 or Google Cloud Storage) and distributed filesystems (such as Ceph, GlusterFs or HDFS). Since this article is about running Spark on Hadoop, we'll persist our dataset to HDFS.

Firstly, raw data will be persisted in order to keep the unaltered input. Secondly, the results of the streaming query will be persisted to avoid recalculations from the raw dataset.

Persisting raw data with Spark Structured Streaming

A Spark Structured Streaming query can be written to the filesystem by providing the URI associated with the `path` property of the `DataFrame.writeStream` instance. Note, the URI scheme determines the type of file storage. Two queries in the code below are launched right after the ingestion and parsing of the data. Streaming DataFrames weren't cleaned, joined, nor processed yet.

```
from pyspark.sql.functions import year, month, dayofmonth

sdfRides.withColumn("year", year("startTime")) \
  .withColumn("month", month("startTime")) \
  .withColumn("day", dayofmonth("startTime")) \
  .writeStream \
  .queryName("Persist the raw data of Taxi Rides") \
  .outputMode("append") \
  .format("parquet") \
  .option("path", "hdfs//namenode:namenode-port/tmp/datalake/RidesRaw
  .option("checkpointLocation", "hdfs//namenode:namenode-port/tmp/che
  .partitionBy("year", "month", "day") \
  .option("truncate", False) \
  .start()

sdfFares.withColumn("year", year("startTime")) \
  .withColumn("month", month("startTime")) \
  .withColumn("day", dayofmonth("startTime")) \
  .writeStream \
  .queryName("Persist the raw data of Taxi Fares") \
  .outputMode("append") \
  .format("parquet") \
  .option("path", "hdfs//namenode:namenode-port/tmp/datalake/FaresRaw
  .option("checkpointLocation", "hdfs//namenode:namenode-port/tmp/che
  .partitionBy("year", "month", "day") \
  .option("truncate", False) \
  .start()
```

The chosen file format, Parquet, is a column-oriented data storage format which provides effective storage and processing optimizations. Other file formats could be more appropriate depending on the cases. A comparable alternative to Parquet is the ORC file format which offers complete support for Hive transactional tables with ACID properties. Starting with Spark 2.4, the popular Apache Avro data serialization format is also supported as a built-in data source. Otherwise, traditional file formats such as csv and json are supported.

The `path` option is the URI of the Hadoop directory where the results shall be stored. The NameNode address and port can be found in the *core-site.xml* configuration file. The `fs.defaultFS` property provides the NameNode's host and the port.

Spark's file sink supports writes to partitioned tables. In the code above, the DataFrames were partitioned according to the year, month, and the day. Thus, data is persisted accordingly to the directory structure "year/month/day". For example, a file would be stored on HDFS as "tmp/datalake/RidesRaw/year=2013/month=1/day=1/part-00000-1031a1c0-8a0c-4b47-ae46-a81615f0a607.c000.snappy.parquet". Notice that the partitioning was accomplished by extracting the hour, month, and the day from the Taxi ride event timestamp. New columns storing year, month, and the day had to be created to enable chosen partitioning.

Spark Structured Streaming provides fault-tolerance with checkpointing. Streaming state `checkpointLocation` must be specified for recovery. If you are not familiar with checkpointing, it is defined as the process of truncating RDD lineage graph and saving it to a reliable distributed (HDFS) or local file system. In Spark Structured Streaming, it persists the state of streaming metadata to HDFS in case of failures. For instance, if an executor goes down, Spark can recover the last offset from where to read the Kafka topic.

The methodology in this section implies that a single message received from Kafka topic is persisted only once on HDFS, without duplicates. Note that each Taxi ride entails 3 types of events: the beginning of the ride, the end of the ride, and the fares information that are collected seperately when the ride ends.

Writing Spark streaming query results to the filesystem

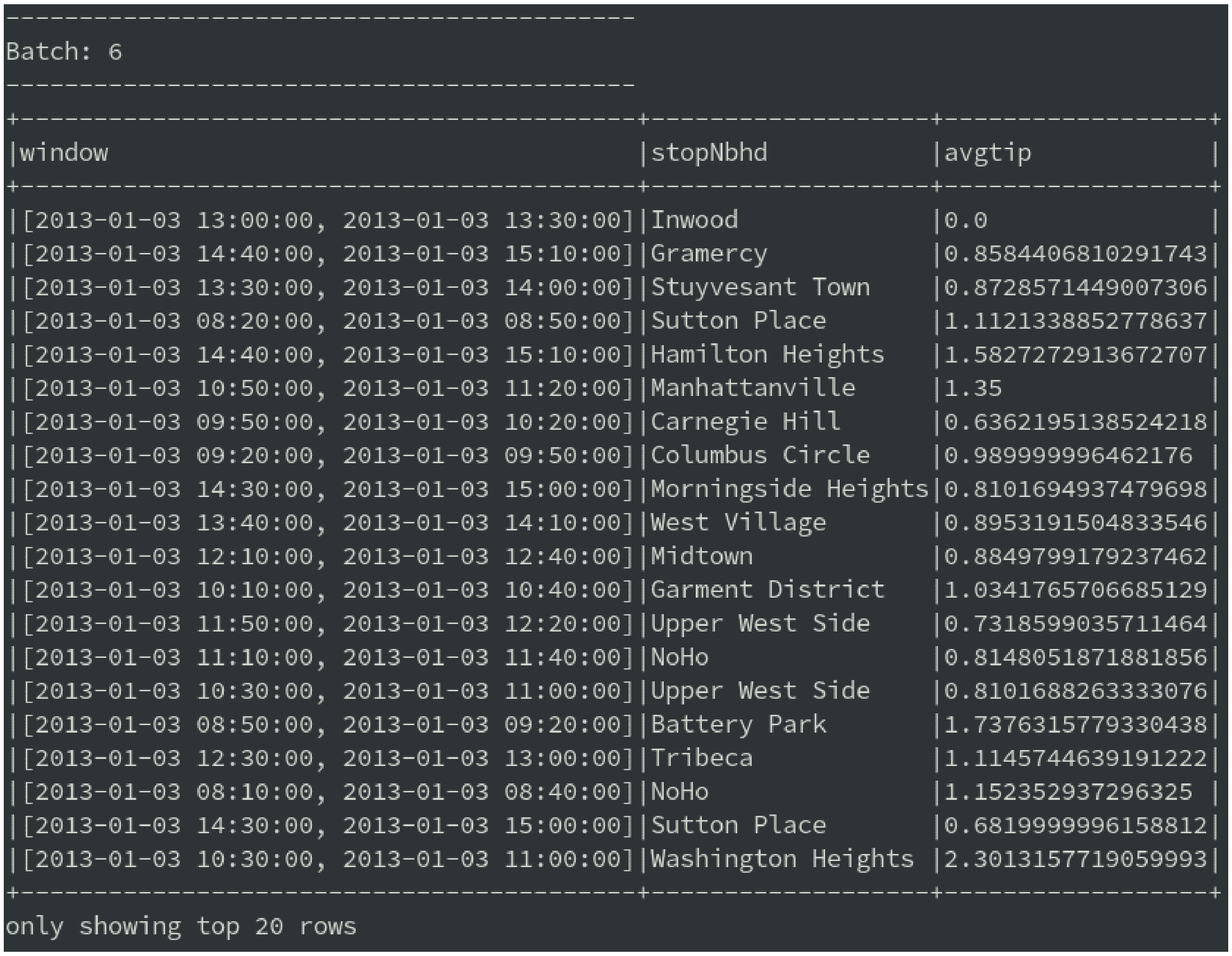
In addition to the persistence of the raw data, the results of the Spark streaming query can be persisted:

```
tips.writeStream \
  .queryName("Persist the processed data") \
  .outputMode("append") \
  .format("parquet") \
  .option("path", "hdfs//namenode:namenode-port/tmp/datalake/taxiTips
  .option("checkpointLocation", "hdfs//namenode:namenode-port/tmp/che
  .partitionBy("window") \
  .option("truncate", False) \
  .start() \
  .awaitTermination()
```

The “tips” streaming DataFrame has been already processed, as showed the first part. In the code above, the partitioning is specified on the “window” column. Recall that this column was obtained in the “Adding aggregation” section of the first part. The aggregation on the tip column was done in the very last stage of processing:

```
tips = sdf \
  .groupBy(
    window("endTime", "30 minutes", "10 minutes"),
    "stopNbhd") \
  .agg(avg("tip"))
```

The aggregate query is calculating the average tip for each neighborhood on 30 minutes time windows. New windows start at 10 minutes intervals, thus some windows are overlapping. The output of the streaming query in the console was looking as below:



```
-------------------------------------------
Batch: 6
-------------------------------------------
+------------------------------------------+--------------------+------------------+
|window                                    |stopNbhd            |avgtip            |
+------------------------------------------+--------------------+------------------+
|[2013-01-03 13:00:00, 2013-01-03 13:30:00]|Inwood              |0.0               |
|[2013-01-03 14:40:00, 2013-01-03 15:10:00]|Gramercy            |0.858440681029174З|
|[2013-01-03 13:30:00, 2013-01-03 14:00:00]|Stuyvesant Town     |0.8728571449007306|
|[2013-01-03 08:20:00, 2013-01-03 08:50:00]|Sutton Place        |1.1121338852778637|
|[2013-01-03 14:40:00, 2013-01-03 15:10:00]|Hamilton Heights    |1.5827272913672707|
|[2013-01-03 10:50:00, 2013-01-03 11:20:00]|Manhattanville      |1.35              |
|[2013-01-03 09:50:00, 2013-01-03 10:20:00]|Carnegie Hill       |0.6362195138524218|
|[2013-01-03 09:20:00, 2013-01-03 09:50:00]|Columbus Circle     |0.989999996462176 |
|[2013-01-03 14:30:00, 2013-01-03 15:00:00]|Morningside Heights|0.8101694937479698|
|[2013-01-03 13:40:00, 2013-01-03 14:10:00]|West Village        |0.8953191504833546|
|[2013-01-03 12:10:00, 2013-01-03 12:40:00]|Midtown             |0.8849799179237462|
|[2013-01-03 10:10:00, 2013-01-03 10:40:00]|Garment District    |1.0341765706685129|
|[2013-01-03 11:50:00, 2013-01-03 12:20:00]|Upper West Side     |0.7318599035711464|
|[2013-01-03 11:10:00, 2013-01-03 11:40:00]|NoHo                |0.8148051871881856|
|[2013-01-03 10:30:00, 2013-01-03 11:00:00]|Upper West Side     |0.8101688263333076|
|[2013-01-03 08:50:00, 2013-01-03 09:20:00]|Battery Park        |1.7376315779330438|
|[2013-01-03 12:30:00, 2013-01-03 13:00:00]|Tribeca             |1.1145744639191222|
|[2013-01-03 08:10:00, 2013-01-03 08:40:00]|NoHo                |1.15235293729632 5 |
|[2013-01-03 14:30:00, 2013-01-03 15:00:00]|Sutton Place        |0.6819999996158812|
|[2013-01-03 10:30:00, 2013-01-03 11:00:00]|Washington Heights  |2.3013157719059993|
+------------------------------------------+--------------------+------------------+
only showing top 20 rows
```

Results are written to the disk according to the directory structure specified by partitioning. Partitioning is specified on the “window” column, hence each record in the table above will be written to a separate directory, named after the record’s time window. For example, the window starting at 9:50 and ending at 10:20 on 2013/01/01 has a directory `/tmp/datalake/window=[2013-01-0109:50:00,2013-01-0110:20:00]`. Numerous files belonging to the window are stored within each window directory, e.g. `part-00008-cedf7585-be4c-469b-80f6-5cc5a0c4623e.c000.snappy.parquet`.

The 30-minutes long aggregate windows slide in 10 minutes intervals. Hence, a single Taxi ride can be used for calculation up to 3 times. A single Taxi ride influences many windows, but each window has a unique set of average tips calculated for neighborhoods.

Writing Spark streaming query results to multiple locations

The method for persistent storage of streaming query results on HDFS was demonstrated. We wish to keep both types of storage for results: persistent in HDFS and low-latency in-memory. Unfortunately, the `writeStream` interface is designed to support only one destination for the results of the streaming query. Although the stream processing results cannot be simply duplicated to another sink, other strategies can be employed to accomplish data storage both in-memory and in a distributed filesystem.

- It is possible to launch 2 streaming queries in parallel inside a Spark application. The first query performs the processing and mounts the final results in memory, while the second query periodically reads them from there and writes to persistent storage
- Instead of running queries in a single application, they could be assigned to 2 individual Spark applications consuming the same Kafka topic.
- Another solution could be using the `StreamSinkProvider` available from Spark 2 to implement a custom Sink
- Finally, the `foreachBatch` sink available from Spark 2.4 can be leveraged to implement custom logic that writes query results to multiple locations

The choice of the solution depends on the use case, latency and performance requirements, budget, existing data platform, and other factors.

Compaction of HDFS files

The Parquet files persisted on HDFS are small files grouped into directories corresponding to partitioning (e.g. aggregate windows). The overhead for storing large amount of small files is high because HDFS store all the metadata of the filesystem in memory.

Thus, storing too many small files is considered a bad practice. It is however acceptable if only a few of them are stored temporarily. The stream created above wouldn’t cause issues until there are millions or even billions of small files accumulated. A periodical compaction batch job could be created to merge them into larger ones. A batch job written in Spark would need to enumerate directories corresponding to each time window and merge files to e.g. daily chunks. That batch job could be scheduled daily with a tool like Apache Oozie or Apache Airflow.

Monitoring Spark applications

Monitoring Spark applications and alerting upon collection of abnormal metrics is necessary to prevent applications’ malfunctioning and breakdowns in a production environment. Monitoring is also invaluable to measure and evaluate the performance of applications. The monitoring, collection, visualization, and alerting of a Spark application on YARN will be considered.

Generally, the state of the whole cluster has to be monitored, as its health affects the performance of applications running on it. Cluster monitoring won't be covered here. It could be achieved with the tools available in Hadoop distributions and/or completed with a custom collection of metrics and logs. For instance, the HDP distribution leverages Apache Ambari which comes with a metric system, HTTP dashboards and an alert framework.

Which metrics to monitor

Hadoop YARN allocates resources for Spark application by assigning YARN resource containers to it. Thus, monitoring the job information inside YARN is essential to the monitoring of Spark applications scheduled by YARN. It contains multiple metrics related to the job as well as to each task scheduled inside it. This includes the start/end time, the status, the configuration, … Other important indicators are related to the application's utilization of resources. For example, one could profile the performance of Spark jobs by checking if the application is bounded by its RAM, or if the number of CPU cores is a bottleneck. These resources are managed by YARN's ResourceManager and corresponding metrics can be monitored in YARN.

To examine more detailed information about Spark application and its execution, Spark itself can be monitored. Spark application metrics describe the Spark's driver, executors, and other Spark components. Metrics about scheduling and execution of the jobs, stages, and tasks are especially useful for performance troubleshooting of a Spark application. Other examples are RDD sizes, memory usage for caching data, and when an executor is removed.

Collecting metrics of the Spark application

All the metrics are available through the YARN's and Spark's web services. Web UIs provide a convenient access and are completed with REST API end-points for automation retrieval.

Prometheus is as a modern monitoring system. Prometheus can be configured to fetch metrics from targets specified as HTTP resource paths. Metrics from third-party systems can be exported as Prometheus metrics. Especially useful here is the JMX exporter which exports metrics from JVM-based applications. Both Spark and YARN could be configured to use the Java Management Extensions (JMX) as a sink. JMX end-point could be in turn used by Prometheus to scrap all the metrics automatically. To enable metrics of Spark Structured Streaming queries, set the spark.sql.streaming.metricsEnabled property to true.

Alerting on metrics

The monitoring system now has all the metrics collected in Prometheus. Inspecting them manually isn't feasible for numerous applications and metrics. The monitoring system needs alerting capabilities to truly benefit from the collected metrics. This approach is more reliable, it frees the administrators from daunting tracking of metrics, and more metrics could be monitored without a cost increase.

For example, one could setup the alerting framework to notify cluster administrators when the streaming job has failed and the service is unavailable. Ideally, an automatic notification system sends information about suspected metrics before the application crashes. Alerts should be set on resource and performance metrics to detect if a value exceeds a certain threshold (or is below it).

Visualizing metrics in dashboards

Prometheus integrates with Grafana to provide visualizations and dashboards of the collected metrics. Visual examination of graphed metrics could provide insights into the temporal behavior of an application. Visualizations and dashboards are especially useful to investigate problems reported by the alerting framework. With dashboards, an administrator can trace abnormalities in visual plots of the reported metric. For example, upon inspection of the reported peak, it could turn out to be an expected behavior of the application due to seasonality. Other times, a metric could indicate a serious problem with a Spark application.

Spark Structured Streaming reliability

Another important aspect of a production environment is the stability of the service. Spark Structured Streaming is improving with each release and is mature enough to be used in production. However, like most of the software, it isn't bug-free. For example, Spark Structured Streaming in `append` mode could result in missing data (SPARK-26167). For the cases with features like S3 storage and stream-stream join, "append mode" is required. Even if it was resolved in Spark 2.4 (SPARK-24156), most users won't be able to benefit from the fix yet. For example, the latest HDP 3.1.0 still ships with Spark 2.3. Many companies still have HDP 2.x and older Spark versions. Given a recent merger between Cloudera and Hortonworks, the recent version of Spark likely won't be available before the release of the Cloudera Data Platform (CDP). New Spark Structured Streaming features from Spark 2.3 onward are ought to be introduced with care since bug fixes won't be available right away.

Security on a cluster

In this article, the test cluster running the streaming application isn't secured. In a real production environment, there are numerous security considerations: authentication (LDAP, Kerberos, SSL/TLS, login/password), authorization (RBAC, ABAC, RAdAC etc.), network policies (Firewall, VLAN isolation, NAT rules), encryption of data (in transit/at rest), and data governance. For example, in an HDP environnement, the following services are usually deployed on a cluster to accommodate security needs:

- Apache KNOX leverages Kerberos authentication protocol and network policies into perimeter security on a Hadoop cluster
- Apache Ranger leverages authorization access control models into standardized authorization across a Hadoop cluster
- Apache Atlas provides metadata driven governance for Hadoop cluster

Summary

To reap the real benefits of the Spark processing, it must run in a distributed environment such as a multi-node Hadoop cluster. The development of a Spark Structured Streaming application is only one part of the work. Deploying, securing, and monitoring a Spark application inside an Hadoop cluster is a process which imply programming, data confort, devops techniques, and infrastructure understanding, to name a few. Accounting for discrepancies and dependencies between components is cubersome. Moreover, launching and maintaining Spark Structured Streaming applications on a cluster is more difficult than on a local machine because the stream processing jobs need to be tuned for the cluster and monitored well.

Share this article