TIBCOSoftware / **snappy-examples** Public

Use cases built on SnappyData. Use cases contained here: 1. Ad Analytics 2. Streaming data ingestion from RabbitMQ.

32 stars 17 forks

Star Watch

Code Issues Pull requests 2 Actions Projects Security Insight

master

sumwale filtered-backup: update SnappyData to 1.3.1 … on Jul 18, 2022 118

View code

README.md

We benchmarked this code example against the MemSQL Spark Connector and the Cassandra Spark Connector. SnappyData outperformed Cassandra by 45x and MemSQL by 3x on query execution while concurrently ingesting. The benchmark is described here.

There is a screencast associated with this repo here

Skip directly to instructions

Table of Contents

Introduction

SnappyData aims to deliver real time operational analytics at interactive speeds with commodity infrastructure and far less complexity than today. SnappyData fulfills this promise by

- Enabling streaming, transactions and interactive analytics in a single unifying system rather than stitching different solutions—and
- Delivering true interactive speeds via a state-of-the-art approximate query engine that leverages a multitude of synopses as well as the full dataset. SnappyData implements this by deeply integrating an in-memory database into Apache Spark.

One out-of-place "filtered-backup" project has been added separately that adds a job which allows for a backup with exclusion/inclusion patterns for disk stores to exclude/include -- see the README.txt in that directory.

Purpose

Here we use a simplified Ad Analytics example, which streams in AdImpression logs, pre-aggregating the logs and ingesting into the built-in in-memory columnar store (where the data is stored both in 'exact' form as well as a stratified sample). We showcase the following aspects of this unified cluster:

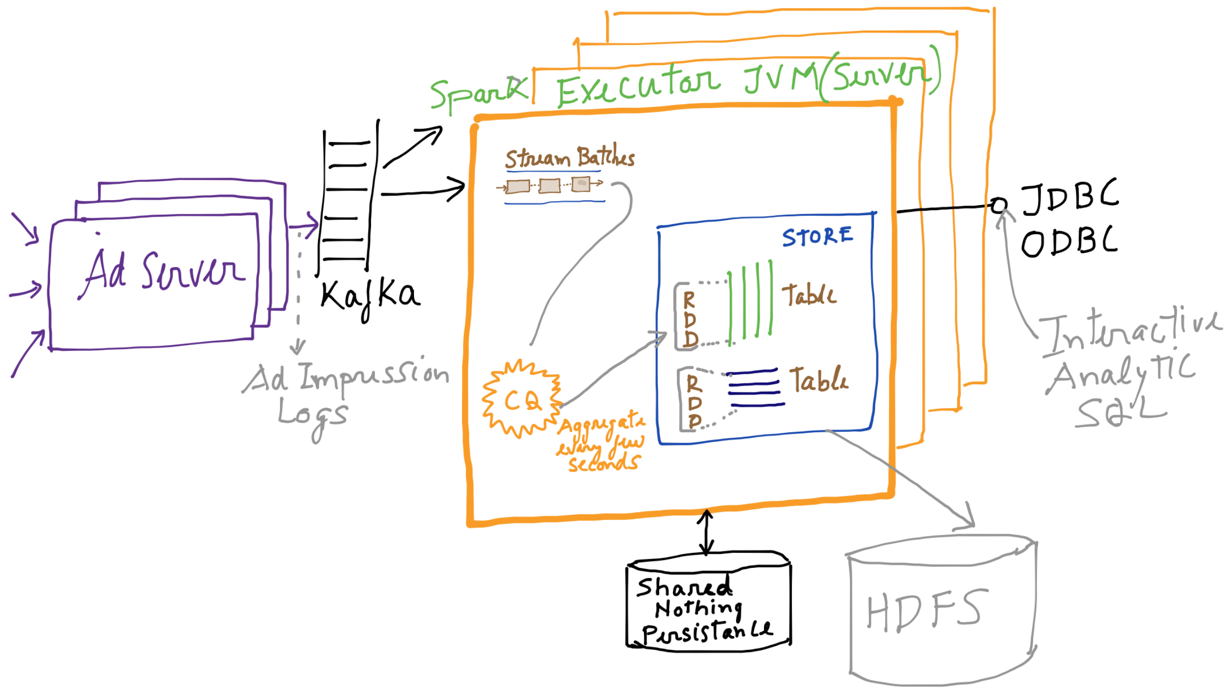
- Simplicity of using SQL or the DataFrame API to model streams in spark.
- The use of SQL/SchemaDStream API (as continuous queries) to pre-aggregate AdImpression logs (it is faster and much more convenient to incorporate more complex analytics, rather than using map-reduce).
- Demonstrate storing the pre-aggregated logs into the SnappyData columnar store with high efficiency. While the store itself provides a rich set of features like hybrid row+column store, eager replication, WAN replicas, HA, choice of memory-only, HDFS, native disk persistence, eviction, etc we only work with a column table in this simple example.
- Run OLAP queries from any SQL client both on the full data set as well as sampled data (showcasing sub-second interactive query speeds). The stratified sample allows us to manage an infinitely growing data set at a fraction of the cost otherwise required.

Ad Impression Analytics use case

We borrow our use case implementation from this blog - We more or less use the same data structure and aggregation logic and we have adapted this code to showcase the SnappyData programming model extensions to Spark. We retain the native Spark example for comparison.

Our architecture is depicted in the figure below.

We consider an adnetwork where adservers log impressions in Apache Kafka (distributed publish-subscribe messaging system). These impressions are then aggregated by Spark Streaming into the SnappyData Store. External clients connect to the same cluster using JDBC/ODBC and run arbitrary OLAP queries. As AdServers can feed logs from many websites and given that each AdImpression log message represents a single Ad viewed by a user, one can expect thousands of messages every second. It is crucial that ingestion logic keeps up with the stream. To accomplish this, SnappyData collocates the store partitions with partitions created by Spark Streaming. i.e. a batch of data from the stream in each Spark executor is transformed into a compressed column batch and stored in the same JVM, avoiding redundant shuffles (except for HA).



The incoming AdImpression log is formatted as depicted below.

timestamp	publisher	advertiser	website	geo	bid	
2016-05-25 16:45:29.027	publisher44	advertiser11	website233	NJ	0.857122	c
2016-05-25 16:45:29.027	publisher31	advertiser18	website642	WV	0.211305	c
2016-05-25 16:45:29.027	publisher21	advertiser27	website966	ND	0.539119	c

timestamp	publisher	advertiser	website	geo	bid	
2016-05-25 16:45:29.027	publisher34	advertiser11	website284	WV	0.050856	c
2016-05-25 16:45:29.027	publisher29	advertiser29	website836	WA	0.896101	c

We pre-aggregate these logs by publisher and geo, and compute the average bid, the number of impressions and the number of uniques (the number of unique users that viewed the Ad) every 2 seconds. We want to maintain the last day's worth of data in memory for interactive analytics from external clients. Some examples of interactive queries:

- **Find total uniques for a certain AD grouped on geography;**
- **Impression trends for advertisers over time;**
- **Top ads based on uniques count for each Geo.**

So the aggregation will look something like:

timestamp	publisher	geo	avg_bid	imps	uniques
2016-05-25 16:45:01.026	publisher10	UT	0.5725387931435979	30	26
2016-05-25 16:44:56.21	publisher43	VA	0.5682680168342149	22	20
2016-05-25 16:44:59.024	publisher19	OH	0.5619481767564926	5	5
2016-05-25 16:44:52.985	publisher11	VA	0.4920346523303594	28	21
2016-05-25 16:44:56.803	publisher38	WI	0.4585381957119518	40	31

Code highlights

We implemented the ingestion logic using 3 methods mentioned below but only describe the SQL approach for brevity here.

- Vanilla Spark API (from the original blog).

- Spark API with Snappy extensions to work with the stream as a sequence of DataFrames. (btw, SQL based access to streams is also the theme behind Structured streaming being introduced in Spark 2.0)
- SQL based - described below.

Generating the AdImpression logs

A KafkaAdImpressionGenerator simulates Adservers and generates random AdImpressionLogs(Avro formatted objects) in batches to Kafka.

```
val props = new Properties()
props.put("serializer.class", "io.snappydata.adanalytics.AdImpressionLogAvroE
props.put("partitioner.class", "kafka.producer.DefaultPartitioner")
props.put("key.serializer.class", "kafka.serializer.StringEncoder")
props.put("metadata.broker.list", brokerList)
val config = new ProducerConfig(props)
val producer = new Producer[String, AdImpressionLog](config)
sendToKafka(generateAdImpression())

def generateAdImpression(): AdImpressionLog = {
  val random = new Random()
  val timestamp = System.currentTimeMillis()
  val publisher = Publishers(random.nextInt(NumPublishers))
  val advertiser = Advertisers(random.nextInt(NumAdvertisers))
  val website = s"website_${random.nextInt(Constants.NumWebsites)}.com"
  val cookie = s"cookie_${random.nextInt(Constants.NumCookies)}"
  val geo = Geos(random.nextInt(Geos.size))
  val bid = math.abs(random.nextDouble()) % 1
  val log = new AdImpressionLog()
}

def sendToKafka(log: AdImpressionLog) = {
  producer.send(new KeyedMessage[String, AdImpressionLog](
    Constants.kafkaTopic, log.getTimestamp.toString, log))
}
```

Spark stream as SQL table and Continuous query

SnappySQLLogAggregator creates a stream over the Kafka source. The messages are converted to Row objects using AdImpressionToRowsConverter to comply with the schema defined in the 'create stream table' below. This is mostly just a SQL veneer over Spark Streaming. The stream table is also automatically registered with the SnappyData catalog so external clients can access this stream as a table.

Next, a continuous query is registered on the stream table that is used to create the aggregations we spoke about above. The query aggregates metrics for each publisher and geo every 1 second. This query runs every time a batch is emitted. It returns a SchemaDStream.

```
val sc = new SparkContext(sparkConf)
val snsc = new SnappyStreamingContext(sc, batchDuration)

/**
* AdImpressionStream presents the stream as a Table. It is registered with
* Underneath the covers, this is an abstraction over a DStream. DStream bat
*/
snsc.sql("create stream table adImpressionStream (" +
  " time_stamp timestamp," +
  " publisher string," +
  " advertiser string," +
  " website string," +
  " geo string," +
  " bid double," +
  " cookie string) " +
  " using directkafka_stream options" +
  " (storagelevel 'MEMORY_AND_DISK_SER_2'," +
  " rowConverter 'io.snappydata.adanalytics.AdImpressionToRowsConverter' ,"
  s" kafkaParams 'metadata.broker.list->$brokerList'," +
  s" topics '$kafkaTopic'," +
  " K 'java.lang.String'," +
  " V 'io.snappydata.adanalytics.AdImpressionLog', " +
  " KD 'kafka.serializer.StringDecoder', " +
  " VD 'io.snappydata.adanalytics.AdImpressionLogAvroDecoder')")

  // Aggregate metrics for each publisher, geo every few seconds. Just 1 se
  // With the stream registered as a table, we can execute arbitrary querie
  // These queries run each time a batch is emitted by the stream. A contir
  val resultStream: SchemaDStream = snsc.registerCQ(
    "select min(time_stamp), publisher, geo, avg(bid) as avg_bid," +
      " count(*) as imps , count(distinct(cookie)) as uniques" +
      " from adImpressionStream window (duration 1 seconds, slide 1 seconds
      " where geo != 'unknown' group by publisher, geo")
```

Ingesting into Column table

Next, create the Column table and ingest result of continuous query of aggregating AdImpressionLogs. Here we use the Spark Data Source API to write to the aggrAdImpressions table. This will automatically localize the partitions in the data store without shuffling the data.

```
    snsc.sql("create table aggrAdImpressions(time_stamp timestamp, publisher s
     " geo string, avg_bid double, imps long, uniques long) " +
      "using column options(buckets '11')")
    //Simple in-memory partitioned, columnar table with 11 partitions.
    //Other table types, options to replicate, persist, overflow, etc are defi
    // here -> http://snappydatainc.github.io/snappydata/rowAndColumnTables/

    //Persist using the Spark DataSource API
    resultStream.foreachDataFrame(_.write.insertInto("aggrAdImpressions"))
```

Ingesting into a Sample table

Finally, create a sample table that ingests from the column table specified above. This is the table that approximate queries will execute over. Here we create a query column set on the 'geo' column, specify how large of a sample we want relative to the column table (3%) and specify which table to ingest from:

```
  snsc.sql("CREATE SAMPLE TABLE sampledAdImpressions" +
    " OPTIONS(qcs 'geo', fraction '0.03', strataReservoirSize '50', baseTable
```

Let's get this going

In order to run this example, we need to install the following:

1. Apache Kafka 2.11-0.8.2.1
2. SnappyData 1.0.0 Enterprise Release. Download the binary snappydata-1.0.0-bin.tar.gz and Unzip it. The binaries will be inside "snappydata-1.0.0-bin" directory.
3. JDK 8

Then checkout the Ad analytics example

```
git clone https://github.com/SnappyDataInc/snappy-poc.git
```

Note that the instructions for kafka configuration below are for 2.11-0.8.2.1 version of Kafka.

To setup kafka cluster, start Zookeeper first from the root kafka folder with default zookeeper.properties:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

Start one Kafka broker with default properties:

```
bin/kafka-server-start.sh config/server.properties
```

From the root kafka folder, Create a topic "adImpressionsTopic":

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --partitions 8 --
topic adImpressionsTopic --replication-factor=1
```

Next from the checkout `/snappy-poc/` directory, build the example

```
-- Build and create a jar having all dependencies in assembly/build/libs
./gradlew assemble

-- If you use IntelliJ as your IDE, you can generate the project files
using
./gradlew idea    (Try ./gradlew tasks for a list of all available tasks)
```

Goto the SnappyData product install home directory. In conf subdirectory, create file "spark-env.sh"(copy spark-env.sh.template) and add this line ...

```
SPARK_DIST_CLASSPATH=SNAPPY_POC_HOME/assembly/build/libs/snappy-poc-1.0.0-
assembly.jar
```

> Make sure you set the SNAPPY_POC_HOME directory appropriately above

Leave this file open as you will copy/paste the path for SNAPPY_POC_HOME shortly.

Start SnappyData cluster using following command from installation directory.

```
./sbin/snappy-start-all.sh
```

This will start one locator, one server and a lead node. You can understand the roles of these nodes here

Submit the streaming job to the cluster and start it (consume the stream, aggregate and store).

> Make sure you copy/paste the SNAPPY_POC_HOME path from above in the command below where indicated

```
./bin/snappy-job.sh submit --lead localhost:8090 --app-name AdAnalytics --
class io.snappydata.adanalytics.SnappySQLLogAggregatorJob --app-jar
SNAPPY_POC_HOME/assembly/build/libs/snappy-poc-1.0.0-assembly.jar --stream
```

SnappyData supports "Managed Spark Drivers" by running these in Lead nodes. So, if the driver were to fail, it can automatically re-start on a standby node. While the Lead node starts the streaming job, the actual work of parallel processing from kafka, etc is done in the SnappyData servers. Servers execute Spark Executors collocated with the data.

Start generating and publishing logs to Kafka from the `/snappy-poc/` folder

```
./gradlew generateAdImpressions
```

You can see the Spark streaming processing batches of data once every second in the Spark console. It is important that our stream processing keeps up with the input rate. So, we note that the 'Scheduling Delay' doesn't keep increasing and 'Processing time' remains less than a second.

Next, interact with the data. Fast.

Now, we can run some interactive analytic queries on the pre-aggregated data. From the root SnappyData folder, enter:

```
./bin/snappy-shell
```

Once this loads, connect to your running local cluster with:

```
connect client 'localhost:1527';
```

Set Spark shuffle partitions low since we don't have a lot of data; you can optionally view the members of the cluster as well:

```
set spark.sql.shuffle.partitions=7;
show members;
```

Let's do a quick count to make sure we have the ingested data:

```
select count(*) from aggrAdImpressions;
```

Let's also directly query the stream using SQL:

```
select count(*) from adImpressionStream;
```

Now, lets run some OLAP queries on the column table of exact data. First, lets find the top 20 geographies with the most ad impressions:

```
select count(*) AS adCount, geo from aggrAdImpressions group by geo order by
```

Next, let's find the total uniques for a given ad, grouped by geography:

```
select sum(uniques) AS totalUniques, geo from aggrAdImpressions where publish
```

Now that we've seen some standard OLAP queries over the exact data, let's execute the same queries on our sample tables using SnappyData's Approximate Query Processing techinques. In most production situations, the latency difference here would be significant because the volume of data in the exact table would be much higher than the sample tables. Since this is an example, there will not be a significant difference; we are showcasing how easy AQP is to use.

We are asking for an error rate of 20% or below and a confidence interval of 0.95 (note the last two clauses on the query). The addition of these last two clauses route the query to the sample table despite the exact table being in the FROM clause. If the error rate exceeds 20% an exception will be produced:

```
select count(*) AS adCount, geo from aggrAdImpressions group by geo order by
```

And the second query from above:

```
select sum(uniques) AS totalUniques, geo from aggrAdImpressions where publish
```

Note that you can still query the sample table without specifying error and confidence clauses by simply specifying the sample table in the FROM clause:

```
select sum(uniques) AS totalUniques, geo from sampledAdImpressions where publ
```

Now, we check the size of the sample table:

```
select count(*) as sample_cnt from sampledAdImpressions;
```

Finally, stop the SnappyData cluser with:

```
./sbin/snappy-stop-all.sh
```

So, what was the point again?

Hopefully we showed you how simple yet flexible it is to parallely ingest, process using SQL, run continuous queries, store data in column and sample tables and interactively query data. All in a single unified cluster. We will soon release Part B of this exercise - a benchmark of this use case where we compare SnappyData to other alternatives. Coming soon.

Ask questions, start a Discussion

Stackoverflow Slack Gitter Mailing List Reddit JIRA

Source code, docs

SnappyData Source

SnappyData Docs

This Example Source

SnappyData Technical Paper

Releases

16 tags

Packages

No packages published

Contributors 11

Languages

● **Scala** 83.5% ● **Java** 15.1% ● **Shell** 1.4%