sparklife

@carsondial

General Tips!

* Avoid collect() and groupByKey()!

* Try to filter down before performing calculations on a data set

* In order, prefer DataSets, DataFrames, and then RDDs for your data structures. DataSets give you type safety, massive improvements on serializing, *and* access to the Catalyst optimizer!

* To squeeze more performance and less memory usage, consider using arrays rather than case classes or tuples (less overhead in serialization and faster access)

* Use mapPartitions() instead of map() for re-using heavy objects like connections and parsers rather than creating them on a per-item basis

* If you need to create a custom partitioner scheme, RDDs are your thing rather than DataFrames/DataSets!

Broadcasts And Accumulators

* Remember you can't broadcast a RDD or DataFrame!

* Accumulators cannot be relied upon for exact counts (if a task fails part way, the count will not be reset, leading to double counting!)

* sc.register(acc, "Name") will make the accumulator show up in WebUI with 'Name'

SparkSQL

* Large schemas may break Hive and make JSON reflection need to sample more of the data set (consider using a fixed schema to avoid reflection!)

* Take advantage of predicate push-down into your data layers whereever possible

* Parquet ALL THE THINGS!

* Custom UDFs will be slower than inbuilt SparkSQL UDF

Spark Streaming

* spark.streaming.backpressure.enabled = rue (for backpressure support)

* Experiment with blockInterval and batch luration sizes and observe in WebUI whether our application is keeping up with the stream

* Invest in mapWithState tooling and avoid ıpdateStateByKey - will be faster and onsiderably less memory-hungry

Testing

```
val input1 = sc.parallelize(List[(Int, Double)]
  ((1, 1.1), (2, 2.2), (3, 3.3))).toDF
val input2 = sc.parallelize(List[(Int, Double)]
  ((1, 1.2), (2, 2.3), (3, 3.4))).toDF
assertDataFrameApproximateEquals(input1, input2, 0.11)
// equal
```

spark-testing-base FTW!

Links!

Apache Spark: https://spark.apache.org
MapWithState:
https://databricks.com/blog/2016/02/01/faster-stateful-stream-processing-in-apache-spark-streaming.html
Livy: https://github.com/cloudera/livy
Spark-Job-Server:
https://github.com/spark-jobserver/spark-jobserver
Checkpoints:
http://aseigneurin.github.io/2016/05/07/spark-kafka-achieving-zero-data-loss.html
Spark Streaming ATO 2015:
http://www.slideshare.net/ianpointer/all-things-open-spark-storm-where-when
Arbiter: https://github.com/etsy/arbiter
Spark-flame: https://github.com/falloutdurham/spark-flame

Garbage Collection

* G1GC collector is a good fit for Spark
* Use GC logging and WebUI to determine time spent in GC
* Lots of major GC? Increase executor memory or spark.memory.fraction
* Minor GC? Increase GC Eden
* May also need to increase -XX:ConcGCThreads to around 5-20 to speed up background marking

Partitions

* Start with a partition base of 3x cores in cluster
* Increase by 1.5x and continue until you see performance decrease
* Lots of tasks finishing in short times? Reduce partitions
* Persist expensive RDDs/Datasets to disk with MEMORY_AND_DISK

sparklife ops!!!

Sizing

* 3 - 5 cores per executor
* Rough limit 64Gb per executor to avoid long GC
* More executors > Fewer, large executors
* Remember there's YARN/Mesos/OS overhead too!

Log ALL THE THINGS

log as much as you can. Spark, YARN, HDFS, Kafka, OS, etc

Splunk, ELK, Datadog, Honeycomb will be your friend!

Deploying

* Consider a shadow cluster for new versions of applications (yay Kafka!)
* Spark checkpointing is undefined across code changes!
* Consider storing stream offsets in ZooKeeper and recreating state on start-up independently of checkpointing

Metrics

send everything to graphite* with this one trick!

```
val sparkConf = new spark.SparkConf()
 .set("spark.metrics.conf.*.sink.graphite.class",
      "org.apache.spark.metrics.sink.GraphiteSink")
 .set("spark.metrics.conf.*.sink.graphite.host",
       graphiteHostName)
val sc = new spark.SparkContext(sparkConf)
```

* CSV, JSON, JMX, and Console sinks are also available