Pitfalls of Apache Spark at Scale

Cesar Delgado @hpcfarmer
DB Tsai @dbtsai

Spark + AI Summit Europe, London, Oct 4, 2018

Apple Siri Open Source Team

- We're Spark, Hadoop, HBase PMCs / Committers / Contributors
- We're the advocate for open source
- Pushing our internal changes back to the upstreams
- Working with the communities to review pull requests, develop new features and bug fixes

Apple Siri

The world's largest virtual assistant service powering every iPhone, iPad, Mac, Apple TV, Apple Watch, and HomePod

Apple Siri Data

- Machine learning is used to personalize your experience throughout your day
- We believe privacy is a fundamental human right

Apple Siri Scale

- Large amounts of requests, Data Centers all over the world
- Hadoop / Yarn Cluster has thousands of nodes
- HDFS has hundred of PB
- 100's TB of raw event data per day

Siri Data Pipeline

- Downstream data consumers were doing the same expensive query with expensive joins
- Different teams had their own repos, and built their jars - hard to track data lineages
- Raw client request data is tricky to process as it involves deep understanding of business logic

Unified Pipeline

- Single repo for Spark application across the Siri
- Shared business logic code to avoid any discrepancy
- Raw data is cleaned, joined, and transformed into one standardized data model for data consumers to query on

Technical Details about Strongly Typed Data

- Schema of data is checked in as case class, and CI ensures schema changes won't break the jobs
- Deeply nested relational model data with 5 top level columns
- The total fields are around 2k
- Stored in Parquet format partitioned by UTC day
- Data consumers query on the subset of data

Review of APIs of Spark

DataFrame: Relational untyped APIs introduced in Spark 1.3. From Spark 2.0,

```
type DataFrame = Dataset[Row]
```

Dataset: Support all the untyped APIs in DataFrame

\+ typed functional APIs

Review of DataFrame

```
val df: DataFrame = Seq(
  (1, "iPhone", 5),
  (2, "MacBook", 4),
  (3, "iPhone", 3),
  (4, "iPad", 2),
  (5, "appleWatch", 1)
).toDF("userId", "device", “counts")

df.printSchema()
"""
  |root
  ||-- userId: integer (nullable = false)
  ||-- device: string (nullable = true)
  ||-- counts: integer (nullable = false)
  |
""".stripMargin
```

Review of DataFrame

```
df.withColumn("counts", $"counts" + 1)
  .filter($"device" === "iPhone").show()
"""
  |+------+------+------+
  ||userId|device|counts|
  |+------+------+------+
  ||     1|iPhone|     6|
  ||     3|iPhone|     4|
  |+------+------+------+
""".stripMargin
// $"counts" == df("counts")
// via implicit conversion
```

Execution Plan - Dataframe Untyped APIs

```
df.withColumn("counts", $"counts" + 1).filter($"device" === "iPhone").explain(true)
  """
       |== Parsed Logical Plan ==
       |'Filter ('device = iPhone)
       |+- Project [userId#3, device#4, (counts#5 + 1) AS counts#10]
       |   +- Relation[userId#3,device#4,counts#5] parquet
       |
       |== Physical Plan ==
       |*Project [userId#3, device#4, (counts#5 + 1) AS counts#10]
       |+- *Filter (isnotnull(device#4) && (device#4 = iPhone))
       |   +- *FileScan parquet [userId#3,device#4,counts#5]
       |      Batched: true, Format: Parquet,
       |      PartitionFilters: [],
       |      PushedFilters: [IsNotNull(device), EqualTo(device,iPhone)],
       |       ReadSchema: struct<userId:int,device:string,counts:int>
       |
""".stripMargin
```

Review of Dataset

```
case class ErrorEvent(userId: Long, device: String, counts: Long)

val ds = df.as[ErrorEvent]

ds.map(row => ErrorEvent(row.userId, row.device, row.counts + 1))
.filter(row => row.device == "iPhone").show()
"""
   |+------+------+------+
   ||userId|device|counts|
   |+------+------+------+
   ||     1|iPhone|     6|
   ||     3|iPhone|     4|
   |+------+------+------+
""".stripMargin
// It's really easy to put existing Java / Scala code here.
```

Execution Plan - Dataset Typed APIs

```
ds.map { row => ErrorEvent(row.userId, row.device, row.counts + 1) }.filter { row =>
  row.device == "iPhone"
}.explain(true)
"""
   |== Physical Plan ==
   |*SerializeFromObject [
   |  assertnotnull(input[0, com.apple.ErrorEvent, true]).userId AS userId#27L,
   |  assertnotnull(input[0, com.apple.ErrorEvent, true]).device, true) AS device#28,
   |  assertnotnull(input[0, com.apple.ErrorEvent, true]).counts AS counts#29L]
   |+- *Filter <function1>.apply
   |   +- *MapElements <function1>, obj#26: com.apple.ErrorEvent
   |      +- *DeserializeToObject newInstance(class com.apple.ErrorEvent), obj#25:
   |          com.apple.siri.ErrorEvent
   |         +- *FileScan parquet [userId#3,device#4,counts#5]
   |            Batched: true, Format: Parquet,
   |            PartitionFilters: [], PushedFilters: [],
   |            ReadSchema: struct<userId:int,device:string,counts:int>
""".stripMargin
```

Strongly Typed Pipeline

- Typed Dataset is used to guarantee the schema consistency
- Enables Java/Scala interoperability between systems
- Increases Data Scientist productivity

Drawbacks of Strongly Typed Pipeline

- Dataset are slower than Dataframe https://tinyurl.com/dataset-vs-dataframe
- In Dataset, many POJO are created for each row resulting high GC pressure
- Data consumers typically query on subsets of data, but schema pruning and predicate pushdown are not working well in nested fields

In Spark 2.3.1

```
case class FullName(first: String, middle: String, last: String)

case class Contact(id: Int,
                   name: FullName,
                   address: String)


sql("select name.first from contacts").where("name.first = 'Jane'").explain(true)

"""
  |== Physical Plan ==
  |*(1) Project [name#10.first AS first#23]
  |+- *(1) Filter (isnotnull(name#10) && (name#10.first = Jane))
  |   +- *(1) FileScan parquet [name#10] Batched: false, Format: Parquet,
PartitionFilters: [], PushedFilters: [IsNotNull(name)], ReadSchema:
struct<name:struct<first:string,middle:string,last:string>>
  |
""".stripMargin
```

In Spark 2.4 with Schema Pruning

```
"""
  |== Physical Plan ==
  |*(1) Project [name#10.first AS first#23]
  |+- *(1) Filter (isnotnull(name#10) && (name#10.first = Jane))
  |   +- *(1) FileScan parquet [name#10] Batched: false, Format: Parquet,
PartitionFilters: [], PushedFilters: [IsNotNull(name)], ReadSchema:
struct<name:struct<first:string>>

  |
""".stripMargin
```

- [SPARK-4502], [SPARK-25363] Parquet nested column pruning

In Spark 2.4 with Schema Pruning + Predicate Pushdown

```
"""
  |== Physical Plan ==
  |*(1) Project [name#10.first AS first#23]
  |+- *(1) Filter (isnotnull(name#10) && (name#10.first = Jane))
  |   +- *(1) FileScan parquet [name#10] Batched: false, Format: Parquet,
PartitionFilters: [], PushedFilters: [IsNotNull(name), EqualTo(name.first,Jane)],
ReadSchema: struct<name:struct<first:string>>

  |
""".stripMargin
```

- [SPARK-4502], [SPARK-25363] Parquet nested column pruning
- [SPARK-17636] Parquet nested Predicate Pushdown

Production Query - Finding a Needle in a Haystack

Spark 2.3.1

Spark 2.3.1 | Jobs | Stages | Storage | Environment | Executors | SQL | **Spark shell** application UI

Stages for All Jobs

Completed Stages: 3

Completed Stages (3)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	parquet at <console>:26 +details	2018/09/20 20:38:21	1.2 h	72002/72002	7.1 TB	31.8 MB		
1	parquet at <console>:23 +details	2018/09/20 20:37:41	1 s	1/1				
0	Listing leaf files and directories for 6000 paths: hdfs://nameservice1/user/sirimetrics_bot2/uberstream/20180802/part-00000-7264d04a-200e... parquet at <console>:23 +details	2018/09/20 20:37:38	2 s	6000/6000				

Spark 2.4 with [SPARK-4502], [SPARK-25363], and [SPARK-17636]

Spark 2.4.0-SNAPSHOT | Jobs | Stages | Storage | Environment | Executors | SQL | **Spark shell** application UI

Stages for All Jobs

Completed Stages: 3

Completed Stages (3)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	parquet at <console>:26 +details	2018/09/20 20:29:48	3.3 min	72002/72002	840.0 GB	35.0 MB		
1	parquet at <console>:23 +details	2018/09/20 20:29:20	0.9 s	1/1				
0	Listing leaf files and directories for 6000 paths: hdfs://nameservice1/user/sirimetrics_bot2/uberstream/20180802/part-00000-7264d04a-200e... parquet at <console>:23 +details	2018/09/20 20:29:17	2 s	6000/6000				

- 21x faster in wall clock time
- 8x less data being read
- Saving electric bills in many data centers

Future Work

- Use Spark Streaming can be used to aggregate the request data in the edge first to reduce the data movement
- Enhance the Dataset performance by analyzing JVM bytecode and turn closures into Catalyst expressions
- Building a table format on top of Parquet using Spark's Datasource V2 API to tracks individual data files with richer metadata to manage versioning and enhance query performance

Conclusions

With some work, engineering rigor and some optimizations

Spark can run at very large scale in lightning speed

Thank you