Apache Spark for RDBMS Practitioners: How I Learned to Stop Worrying and Love to Scale

Luca Canali, CERN

#SAISDev11
About Luca

• Data Engineer and team lead at CERN
  – Hadoop and Spark service, database services
  – 18+ years of experience with data(base) services
  – Performance, architecture, tools, internals

• Sharing and community
  – Blog, notes, tools, contributions to Apache Spark

@LucaCanaliDB – http://cern.ch/canali
The Large Hadron Collider (LHC)
LHC data processing teams (WLCG) have built custom solutions able to operate at very large scale (data scale and world-wide operations).
Overview of Data at CERN

• **Physics** Data ~ 300 PB -> EB
  
  • “Big Data” on **Spark** and Hadoop ~10 PB
    - Analytics for accelerator controls and logging
    - Monitoring use cases, this includes use of Spark streaming
    - Analytics on aggregated logs
    - Explorations on the use of Spark for high energy physics

• **Relational databases** ~ 2 PB
  - Control systems, metadata for physics data processing, administration and general-purpose
Apache Spark @ CERN

• Spark on **YARN/HDFS**
  • In total ~1850 physical cores and 15 PB capacity
• Spark on **Kubernetes**
  • Deployed on CERN cloud using OpenStack
  • See also, at this conference: “Experience of Running Spark on Kubernetes on OpenStack for High Energy Physics Workloads”

<table>
<thead>
<tr>
<th>Cluster Name</th>
<th>Configuration</th>
<th>Software Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accelerator logging</td>
<td>20 nodes (Cores 480, Mem - 8 TB, Storage – 5 PB, 96GB in SSD)</td>
<td>Spark 2.2.2 – 2.3.1</td>
</tr>
<tr>
<td>General Purpose</td>
<td>48 nodes (Cores – 892, Mem – 7.5TB, Storage – 6 PB)</td>
<td>Spark 2.2.2 – 2.3.1</td>
</tr>
<tr>
<td>Development cluster</td>
<td>14 nodes (Cores – 196, Mem – 768GB, Storage – 2.15 PB)</td>
<td>Spark 2.2.2 – 2.3.1</td>
</tr>
<tr>
<td>ATLAS Event Index</td>
<td>18 nodes (Cores – 288, Mem – 912GB, Storage – 1.29 PB)</td>
<td>Spark 2.2.2 – 2.3.1</td>
</tr>
</tbody>
</table>
Motivations/Outline of the Presentation

• Report **experience** of using Apache **Spark** at CERN DB group
  – Value of Spark for data coming from DBs
• Highlights of **learning** experience

![Diagram showing the flow from DB to Data Engineer, then to Big Data, and finally to Spark logo.](image)
Goals

• Entice more DBAs and RDBMS Devs into entering Spark community
• Focus on added value of scalable platforms and experience on RDBMS/data practices
  – SQL/DataFrame API very valuable for many data workloads
  – Performance instrumentation of the code is key: use metric and tools beyond simple time measurement
Spark, Swiss Army Knife of Big Data

One tool, many uses
• Data extraction and manipulation
• Large ecosystem of data sources
• Engine for distributed computing
• Runs SQL
• Streaming
• Machine Learning
• GraphX
• ..
Problem We Want to Solve

• Running reports on relational DATA
  – DB optimized for transactions and online
  – Reports can overload storage system
• RDBMS specialized
  – for random I/O, CPU, storage and memory
  – HW and SW are optimized at a premium for performance and availability
Offload from RDBMS

- Building **hybrid** transactional and reporting systems is expensive
  - Solution: offload to a system optimized for capacity
  - In our case: move data to Hadoop + SQL-based engine
Spark Read from RDBMS via JDBC

```scala
val df = spark.read.format("jdbc")
  .option("url","jdbc:oracle:thin:@server:port/service")
  .option("driver","oracle.jdbc.driver.OracleDriver")
  .option("dbtable","DBTABLE")
  .option("user","XX").option("password","YY")
  .option("fetchsize",10000)
  .option("sessionInitStatement",preambleSQL)
  .load()
```

- Optional DB-specific Optimizations
  SPARK-21519

```sql
option("sessionInitStatement","""BEGIN execute immediate 'alter session set "_serial_direct_read"=true'; END;'"""")
```
Challenges Reading from RDMBs

- Important amount of CPU consumed by serialization-deserialization transferring to/from RDMS data format
  - Particularly important for tables with short rows
- Need to take care of data partitioning, parallelism of the data extraction queries, etc
  - A few operational challenges
  - Configure parallelism to run at speed
  - But also be careful not to overload the source when reading
Apache Sqoop to Copy Data

- Scoop is optimized to read from RDBMS
- Sqoop connector for Oracle DBs has several key optimizations
- We use it for incremental refresh of data from production RDBMS to HDFS “data lake”
  - Typically we copy latest partition of the data
  - Users run reporting jobs on offloaded data from YARN/HDFS
Sqoop to Copy Data (from Oracle)

Sqoop has a rich choice of **options** to customize the data transfer

```bash
sqoop import \
--connect jdbc:oracle:thin:@server.cern.ch:port/service \
--username .. \n-P \
-Doraoop.chunk.method=ROWID \
--direct \ 
--fetch-size 10000 \ 
--num-mappers XX \ 
--target-dir XXXX/mySqoopDestDir \ 
--table TABLESNAME \ 
--map-column-java FILEID=Integer,JOBID=Integer,CREATIONDATE=String \ 
--compress --compression-codec snappy \ 
--as-parquetfile
```

Oracle-specific optimizations
Streaming and ETL

- Besides traditional ETL also streaming important and used by production use cases

Streaming data into Apache Kafka
Spark and Data Formats

- Data formats are key for performance
- **Columnar** formats are a big boost for many reporting use cases
  - Parquet and ORC
  - Column pruning
  - Easier compression/encoding
- Partitioning and bucketing also important
Data Access Optimizations

Use Apache Parquet or ORC

- Profit of column projection, filter push down, compression and encoding, partition pruning
Parquet Metadata Exploration

Tools for Parquet metadata exploration, useful for learning and troubleshooting

Row count: 2840100
Size in bytes: 154947970
OFFSET: 4

Column chunk size: compressed / un compressed / ratio

Row group 1:
INT32 SNAPPY D0:1 FPO:4 SZ:2149027/558823/2.43 VC:2840100 ENC:PLAIN_DICTIONARY,BIT_PACKED,RLE
INT32 SNAPPY D0:1 FPO:2419031 SZ:504805/160593/1.50 VC:2840100 ENC:PLAIN_DICTIONARY,BIT_PACKED,RLE

Useful tools:
parquet-tools
parquet-reader

Info at: https://github.com/LucaCanali/Miscellaneous/blob/master/Spark_Notes/Tools_Parquet_Diagnostics.md
SQL Now

After moving the data (ETL), add the queries to the new system Spark. **SQL/DataFrame API** is a powerful engine

- Optimized for Parquet (recently improved ORC)
- Partitioning, Filter push down
- Feature rich, code generation, also has CBO (cost based optimizer)
- Note: it’s a rich ecosystem, options are available (we also used Impala and evaluated Presto)
Spark SQL Interfaces: Many Choices

- **PySpark**, spark-shell, Spark jobs
  - in Python/Scala/Java
- **Notebooks**
  - Jupyter
  - Web service for data analysis at CERN swan.cern.ch
- **Thrift server**
Lessons Learned

Reading Parquet is CPU intensive
• Parquet format uses encoding and compression
• Processing Parquet -> hungry of CPU cycles and memory bandwidth
• Test on dual socket CPU
  – Up to 3.4 Gbytes/s reads
Benchmarking Spark SQL

Running benchmarks can be useful (and fun):
• See how the system behaves at scale
• Stress-test new infrastructure
• Experiment with monitoring and troubleshooting tools

TPCDS benchmark
• Popular and easy to set up, can run at scale
• See https://github.com/databricks/spark-sql-perf
Example from TPCDS Benchmark

TPCDS WORKLOAD ON NXCALS - DATA SET SCALE: 10 TB - QUERY SET V1.4
420 CORES (60 EXECUTORS, 7 CORES EACH), EXECUTOR MEMORY 100GB (14 GB PER CORE)
ORACLE JVM (1.8.0_161) - SPARK 2.3.0

Query Execution Time (Latency) in seconds

Query name

#SAISDev11
Lessons Learned from Running TPCDS

Useful for commissioning of new systems

- It has helped us capture some system parameter misconfiguration before prod
- Comparing with other similar systems
- Many of the long-running TPCDS queries
  - Use shuffle intensively
  - Can profit of using SSDs for local dirs (shuffle)
- There are improvements (and regressions) of
  - TPCDS queries across Spark versions
Learning Something New About SQL

Similarities but also differences with RDBMS:
- Dataframe abstraction, vs. database table/view
- SQL or DataFrame operation syntax

// Dataframe operations
val df = spark.read.parquet("..path..")
df.select("id").groupBy("id").count().show()

// SQL
df.createOrReplaceTempView("t1")
spark.sql("select id, count(*) from t1 group by id").show()
..and Something OLD About SQL

Using SQL or DataFrame operations API is equivalent
• Just different ways to express the computation
• Look at execution plans to see what Spark executes
  — Spark Catalyst optimizer will generate the same plan for the 2 examples

== Physical Plan ==
*(2) HashAggregate(keys=[id#98L], functions=[count(1)])
+- Exchange hashpartitioning(id#98L, 200)
  +- *(1) HashAggregate(keys=[id#98L], functions=[partial_count(1)])
Monitoring and Performance Troubleshooting

Performance is key for data processing at scale

- Lessons learned from DB systems:
  - Use **metrics** and time-based profiling
  - It’s a boon to rapidly pin-point bottleneck resources

- Instrumentation and tools are essential
Spark and Monitoring Tools

• Spark instrumentation
  – Web UI
  – REST API
  – Eventlog
  – Executor/Task Metrics
  – Dropwizard metrics library

• Complement with
  – OS tools
  – For large clusters, deploy tools that ease working at cluster-level

• https://spark.apache.org/docs/latest/monitoring.html
WEB UI

- Part of Spark: Info on Jobs, Stages, Executors, Metrics, SQL,..
  - Start with: point web browser driver_host, port 4040
Spark Executor Metrics System

Metrics collected at execution time

Exposed by WEB UI/History server, EventLog, custom tools

https://github.com/ce rndb/sparkMeasure
Spark Executor Metrics System

What is available with Spark Metrics, SPARK-25170

<table>
<thead>
<tr>
<th>Spark Executor Task Metric name</th>
<th>Short description</th>
</tr>
</thead>
<tbody>
<tr>
<td>executorRunTime</td>
<td>Elapsed time the executor spent running this task. This includes time fetching shuffle data. The value is expressed in milliseconds.</td>
</tr>
<tr>
<td>executorCpuTime</td>
<td>CPU time the executor spent running this task. This includes time fetching shuffle data. The value is expressed in nanoseconds.</td>
</tr>
<tr>
<td>executorDeserializeTime</td>
<td>Elapsed time spent to deserialize this task. The value is expressed in milliseconds.</td>
</tr>
<tr>
<td>executorDeserializeCpuTime</td>
<td>CPU time taken on the executor to deserialize this task. The value is expressed in nanoseconds.</td>
</tr>
<tr>
<td>resultSize</td>
<td>The number of bytes this task transmitted back to the driver as the TaskResult.</td>
</tr>
<tr>
<td>jvmGCTime</td>
<td>Elapsed time the JVM spent in garbage collection while executing this task. The value is expressed in milliseconds.</td>
</tr>
<tr>
<td>+ I/O Metrics, Shuffle metrics, …</td>
<td>26 metrics available, see <a href="https://github.com/apache/spark/blob/master/docs/monitoring.md">https://github.com/apache/spark/blob/master/docs/monitoring.md</a></td>
</tr>
</tbody>
</table>
Spark Metrics - Dashboards

Use to produce dashboards to measure online:
• Number of active tasks, JVM metrics, CPU usage SPARK-25228, Spark task metrics SPARK-22190, etc.

Configure: --conf spark.metrics.conf=metrics.properties
More Use Cases: Scale Up!

- Spark proved valuable and flexible
- Ecosystem and tooling in place
- Opens to exploring more use cases and scaling up!
Use Case: Spark for Physics

Spark APIs for Physics data analysis
• POC: processed 1 PB of data in 10 hours

Input: Physics data in CERN/HEP storage

Read into Spark (spark-root connector)

Apply computation at scale (UDF)

Data reduction (for further analysis)

Histograms and plots of interest

See also, at this conference: “HEP Data Processing with Apache Spark” and “CERN’s Next Generation Data Analysis Platform with Apache Spark”
Use Case: Machine Learning at Scale

Input: labeled data and DL models

Feature engineering at scale

Hyperparameter optimization (Random/Grid search)

Distributed model training

Output: particle selector model

Data and models from Physics

AUC = 0.9867
Summary and High-Level View of the Lessons Learned
Transferrable Skills, Spark - DB

Familiar ground coming from RDBMS world
• SQL
• CBO
• Execution plans, hints, joins, grouping, windows, etc
• Partitioning

Performance troubleshooting, instrumentation, SQL optimizations
• Carry over ideas and practices from DBs
Learning Curve Coming from DB

• Manage heterogeneous environment and usage: Python, Scala, Java, batch jobs, notebooks, etc
• Spark is fundamentally a library not a server component like a RDBMS engine
• Understand differences tables vs DataFrame APIs
• Understand data formats
• Understand clustering and environment (YARN, Kubernets, …)
• Understand tooling and utilities in the ecosystem
New Dimensions and Value

- With Spark you can easily scale up your SQL workloads
  - Integrate with many sources (DBs and more) and storage (Cloud storage solutions or Hadoop)
- Run DB and ML workloads and at scale
  - CERN and HEP are developing solutions with Spark for use cases for physics data analysis and for ML/DL at scale
- Quickly growing and open community
  - You will be able to comment, propose, implement features, issues and patches
Conclusions

- Apache **Spark** very useful and versatile
  - for many data workloads, including DB-like workloads

- DBAs and RDBMS Devs
  - Can profit of Spark and ecosystem to **scale up** some of their workloads from DBs and do more!
  - Can bring **experience** and good practices over from DB world
Acknowledgements

• Members of Hadoop and Spark service at CERN and HEP users community, CMS Big Data project, CERN openlab
• Many lessons learned over the years from the RDBMS community, notably www.oaktable.net
• Apache Spark community: helpful discussions and support with PRs
• Links
  – More info: @LucaCanaliDB – http://cern.ch/canali