Combining Druid and Spark: Interactive and Flexible Analytics at Scale
Harish Butani

The data infrastructure space has seen tremendous growth and innovation over the last few years. With the rapid adoption of open source technologies, organizations are now able to process and analyze data at volumes that were unfathomable a decade earlier. However, given the rapid growth of the space, it can be difficult to keep up with all the new systems that have been created, and more difficult to understand what problems a system is great at solving. Two popular open source technologies, Druid and Spark, are often mentioned as viable solutions for large-scale analytics. Spark is a platform for advanced analytics, and Druid excels at low-latency, interactive queries. Although the high level messaging presented by both projects may lead you to believe they are competitors in the same space, the technologies are in fact extremely complementary solutions. By combining the rich query model of Spark with the powerful indexing technology of Druid, we can build a more powerful, flexible, and extremely low latency analytics solution.

Apache Spark

Apache Spark is an in-memory, general-purpose cluster computing system where the programming model is based on a LINQ style API. Spark provides rich APIs in Java, Scala, Python, and R. The fundamental programming abstraction in Spark is the Resilient Distributed Dataset (RDD), a collection of data that can be transformed (mapped, filtered, joined, or reduced). By caching RDDs at the input and output at the various stages of data computations, Spark is able to avoid unnecessary re-computation, and for certain workflows, most notably iterative workflows, this can be have tremendous performance improvements over vanilla MapReduce.

Druid

Druid is a relative newcomer to the data infrastructure space, but has already seen adoption by some major technology players. Druid is a column-oriented distributed data store that is ideal for powering user-facing data applications. Because interactive user-facing applications require very low query latencies, most Druid queries complete in less than a second. Druid gets its speed from the custom format (known as Druid segments) in which data is stored, which is a combination of a traditional column store, fused with ideas from search infrastructure. Druid also supports streaming data ingestion and batch data loads.

Why Combine the Technologies?

Spark's programming model is used to build analytical solutions that combine SQL, machine learning, and graph processing. Spark is a great general system for data processing, and it is often used by organizations for reporting and large-scale and/or complex data manipulation workflows. What Spark isn't necessarily optimized for is interactive queries, where computations complete as fast as a user is able to navigate through the data using an application (think pivot tables in Microsoft Excel and other tools).

Druid is designed for workflows where the output result set of computations is much smaller than the input set, for example when filtering or aggregating data. These workflows are most commonly found in OLAP queries. Low-latency OLAP queries are extremely useful in business intelligence as enterprises leverage OLAP to develop numerous performance management solutions to analyze KPIs around cost, supply, and activity, broken down by different dimensions.

By combining the analytic capabilities with Spark and the OLAP and low latency capabilities of Druid, we gain several key advantages:

1. Complex analytics endeavors often require examining different aspects of data. By introducing Druid to the equation, we gain the ability to slice-and-dice data at varied levels of data granularity. This enables data inspection from an arbitrary number of views.
2. For scenarios where Druid is already deployed, Spark's richer programming interface enables industry standard BI tools such as Tableau and MicroStrategy to be connected to the infrastructure, providing complex visualizations with very fast response times.
3. For any project already built on top of Spark's ecosystem, the addition of Druid provides the ability to greatly accelerate existing workloads.

What is the Benefit? Benchmarking of Druid and Spark

Before diving into the details around integrating Druid and Spark, let's examine how Druid can accelerate Spark queries. For these benchmarks, we used the TPCH 10G benchmark data set. We converted the star schema of the original data set into a flattened (denormalized) transaction table. For Spark, we further processed the data to use a monthly partitioned table, stored in a Parquet format. For Druid, we ingested data with pre-aggregation (roll-up) disabled.

The data set sizes are:

TPCH Flat TSV	46.80GB
Druid Segments in HDFS	17.04GB
TPCH Flat Parquet	11.38GB
TPCH Flat Parquet Partitioned by Month	11.56GB

We picked a mixture of business intelligence queries for our benchmarks: slice-and-dice queries common in OLAP scenarios, and aggregation queries common in reporting workloads. All benchmarks were run on the same hardware. For Druid, we ran Druid Historical servers on the hardware, and for the Spark, we ran Spark Executors on the hardware.

We measured the performance of queries where dimensions were filtered for different time intervals and measures were aggregated for different time slices. We benchmarked TPCH queries Q1, Q3, Q5, Q7 and Q8 and a set of pertinent, custom queries.

Below are the details of the queries benchmarked:

Query	Interval	Filters	Group By	Aggregations
Basic Agg.	None	None	ReturnFlag LineStatus	Count(*) Sum(exdPrice)

				…
Ship Date Range	1995-12/1997-09		ReturnFlag LineStatus	Count(*)
SubQry Nation, PartType, ShipDt Range	1995-12/1997-09	PType S_Nation + C_Nation	S_Nation	Count(*) Sum(exdPrice) …
TPCH Q1	None	None	ReturnFlag LineStatus	Count(*) Sum(exdPrice) …
TPCH Q3	1995-03/-	O_Date MktSegment	OKey ODate ShipPri	Sum(exdPrice)
TPCH Q5	None	O_Date Region	S_Nation	Sum(exdPrice)
TPCH Q7	None	S_Nation + C_Nation	S_Nation C_Nation ShipDate Year	Sum(extPrice)
TPCH Q8	None	Region Type ODate	ODate.Year	Sum(exdPrice)

The custom queries we ran are listed below:

Basic Aggregation:

```
SELECT        l_returnflag,
              l_linestatus,
              Count(*),
              Sum(l_extendedprice) AS s,
              Max(ps_supplycost) AS m,
              Avg(ps_availqty) AS a,
              Count(DISTINCT o_orderkey)
FROM   orderlineitempartsupplier
GROUP BY      l_returnflag,
              l_linestatus
```

Ship Date Range:

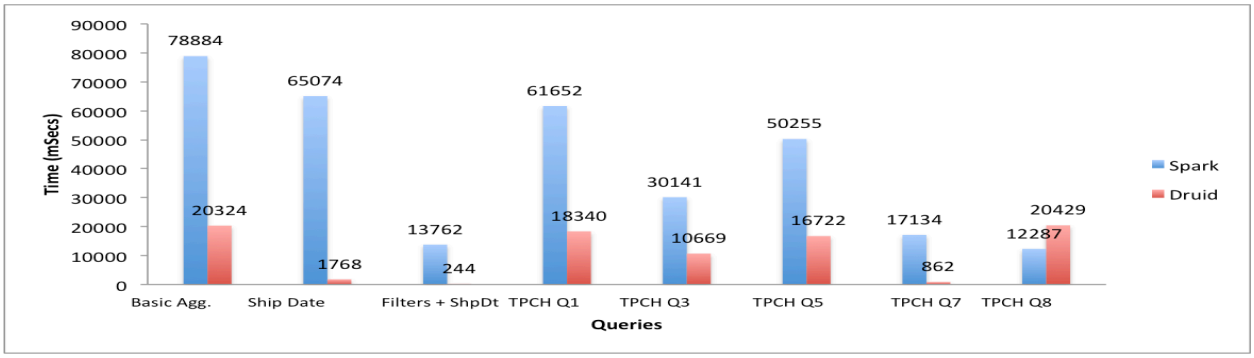
```
SELECT        f,
              s,
              Count(*) AS count_order
FROM   (SELECT       l_returnflag AS f,
                     l_linestatus AS s,
                     l_shipdate,
                     s_region,
                     s_nation,
                     c_nation
              FROM orderlineitempartsupplier) t
WHERE  Dateisbeforeorequal(Datetime(`l_shipdate`),
                     Dateminus(Datetime("1997-12-01"), Period("p90d")))
              AND Dateisafter(Datetime(`l_shipdate`), Datetime("1995-12-01"))
GROUP BY      f,
              s
```

Filters + Ship Date Range:

```
SELECT        s_nation,
              Count(*) AS count_order,
```

```
            Sum(l_extendedprice) AS s,
            Max(ps_supplycost) AS m,
            Avg(ps_availqty) AS a,
            Count(DISTINCT o_orderkey)
FROM   (SELECT      l_returnflag AS f,
                        l_linestatus AS s,
                        l_shipdate,
                        s_region,
                        s_nation,
                        c_nation,
                        p_type,
                        l_extendedprice,
                        ps_supplycost,
                        ps_availqty,
                        o_orderkey
            FROM orderlineitempartsupplier
            WHERE p_type = 'ECONOMY ANODIZED STEEL') t
WHERE  Dateisbeforeorequal(Datetime(`l_shipdate`),
                  Dateminus(Datetime("1997-12-01"), Period("p90d")))
            AND Dateisafter(Datetime(`l_shipdate`),
                  Datetime("1995-12-01"))
            AND    (
                  ( s_nation = 'FRANCE' AND c_nation = 'GERMANY' )
                  OR ( c_nation = 'FRANCE' AND s_nation = 'GERMANY') )
GROUP BY s_nation
```

The full details can be found in our Benchmark Report. Below we list the results of the benchmark:



We can note a few interesting findings from these results:

- The Filters + Ship Date query provides the greatest performance gain (over 50 times over Spark) when Druid is used. This is not surprising as this query is a typical slice-and-dice query tailor-made for Druid. Along the same lines, TPCH Q7 shows a significant performance boost when running on Druid: milliseconds on Druid vs. 10s of seconds on Spark. We can also see that any queries with dimensional predicates and small output result set gain a significant boost when executed in Druid.
- For TPCH Q3, Q5, and Q8 there is an improvement, but not to the same level as Q7. This is because the OrderDate predicate is translated to a JavaScript filter in Druid, which is significantly slower than a native Java filter.
- The Basic Aggregation and TPCH Q1 queries definitely show improvement. The Count-Distinct operation is translated to a cardinality aggregator in Druid, which is an

approximate count. This is definitely an advantage for Druid, especially for large cardinality dimensions. We still need to test this query against Spark using the HyperLogLog aggregator.

We have tried to make the comparisons as fair as possible. We disabled pre-aggregations/roll-ups in Druid (which can compact Druid data sizes 100x and subsequently improve performance 100x), we made sure the Druid segments contained all the data columns, and we disabled query caching in Druid. We also partitioned and cached the Spark DataFrames. Although we can run these experiments with different optimizations such as Code-Generation and Tungsten for Spark, or enabling roll-up and caching in Druid, we are confident that the core result holds: queries that have time partitioning or dimensional predicates (like those commonly found in OLAP workflows) are significantly faster in Druid.

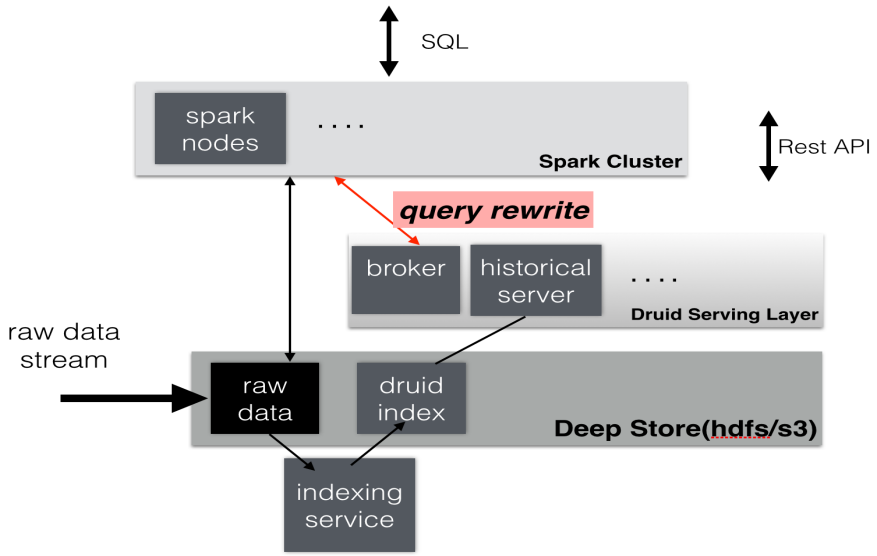
Combining Druid and Spark

Our benchmarks showcase one of the important reasons why the two technologies should be combined. By integrating Druid with Spark, we gain the interactive queries of Druid with the rich programming APIs of Spark.

In terms of how we did the Druid/Spark integration, the key component we used was Spark SQL. Spark SQL enables SQL queries within Spark programs and consists of the DataSources API, the DataFrames data model, and the Catalyst Optimizer to provide a powerful language and translation toolkit. The DataSources API provides a mechanism for Spark SQL to access structured data in external systems, and expose this data as a DataFrame in Spark. A DataFrame is the equivalent of a relational table. DataFrames lie at the core of Spark's extensible data platform. Different language and API builders can plug into this platform by translating their queries down to DataFrames, and lower level features such as optimizations, security enforcements, and data placement policies can all be easily plugged into the system. Since these lower level components are a part of the DataFrames abstraction, they become available to all higher-level components that are built on top of the abstraction.

The overall picture of the integration is the following:

- Raw event data is stored in HDFS or S3, and Druid segments are periodically generated from this data.
- There is a Spark DataSource that has the same schema as the raw event data. This DataSource is also configured with connection information to both the raw data and to the corresponding Druid segments.



We have opened source our work with integrating these two open source systems under the Spark Druid package, and this package consists of 2 main components:

1. DruidDataSource is a Spark Datasource that wraps a DataFrame that exposes the raw data set and also has information about the Druid segments for the data set.
2. During query planning, the DruidPlanner attempts to apply a set of rewrite rules to convert a logical query plan on the raw data set DataFrame into a Druid query.

An example of defining a Druid DataSource is shown below:

```
CREATE TEMPORARY TABLE orderLineItemPartSupplier
      USING org.sparklinedata.druid
      OPTIONS (sourceDataframe "orderLineItemPartSupplierBase",
      timeDimensionColumn "l_shipdate",
      druidDatasource "tpch",
      druidHost "localhost",
      druidPort "8082",
      columnMapping '{  "l_quantity" : "sum_l_quantity",
                        "ps_availqty" : "sum_ps_availqty"
                     }     '
)
```

Consider the 'Filters + Ship Date Range' query listed above. Its logical query plan looks like this:

```
Aggregate [s_nation#88], [s_nation#88,COUNT(1) AS count_order#129L,SUM(l_extendedprice#66) AS ...
 Project [l_extendedprice#66,o_orderkey#53,ps_supplycost#81,s_nation#88,ps_availqty#80]
  Filter ((p_type#93 = ECONOMY ANODIZED STEEL) && ((scalaUDF(scalaUDF(l_shipdate#71),...
   Relation[o_orderkey#53,o_custkey#54,o_orderstatus#55,o_totalprice#56,o_orderdate#57,...
```

The execution of this query plan in Spark involves filtering (and partition pruning) on Part, Nation and Ship Date, and aggregation by Nation.

The rewritten Plan using Druid is:

```
Project [s_nation#88,alias-1#161L AS count_order#129L,alias-2#160 AS s#130,...
 PhysicalRDD [alias-2#160,alias-3#164,...], DruidRDD[8] at RDD at DruidRDD.scala:34
```

The filtering and aggregation is pushed down to the Druid segment. This slice-and-dice query is ideal for Druid's segment storage structure, and as we demonstrated in the benchmarks shown in the previous section, the difference in execution time can be more than an order of magnitude.

Conclusion

Combining Spark and Druid can enable both interactive and flexible analytics at scale. We make this integration happen by leveraging the Spark SQL vision of a plug-in architecture for languages and transformations. The resulting stack is something that combines the low latency OLAP capabilities of Druid with the general-purpose analytic capabilities of Spark. This work is available as an open source Spark package. We hope to build on these ideas to provide even more powerful OLAP analytics in the future, including support for dimensional metadata, MDX, and additional tool connectivity.

About the Author

Harish has over 20 years experience in building backend systems. For the past 10 years he has been focused on data and analytics. Harish is part of the Apache Hive PMC, where his focus was language features like windowing and correlated subqueries and translation/optimizations. He has developed a couple of MDX engines including the MDX engine in v1 of SAP Hana, and the OLAP stack for Cloud9Analytics. Currently he is focused on building richer analytics on Spark.