# Spark and Druid: a suitable match

Harish Butani

September 15, 2015

The World is awash in machine-generated data and the it is growing at a rapid pace [1]. A large portion of this is **Raw Event** data. Event data captures activity (human or machine) at a very fine grained level. Event datasets are also characterized by the recording of large amount of **context** about the Event. It is not uncommon to capture 10s to 100s of contextual Attributes, across many Dimensions: Geography, Time, Customer/User, Product etc. So logically a *Event Dataset* is a very large multi-dimensional Cube that is stored in the form the events are captured: a denormalized/flattened wide table.

What do companies want to do with this data? This data drives a myriad of Analysis like User Targeting, Campaign Attribution and Site Optimization in the Ad. Tech. space; to Smart Cities, Smart Environment, Smart Water Applications in the IOT space [2]. To solve these problems they architect solutions that contain many of these components: a Data collection component(Apache Kafka or Flume), a Stream Processing Layer(Apache Storm. Samza or Spark Streaming), a Storage Layer(HDFS or S3), a Batch Processing Layer(Apache Hive/Tez, Spark, Cascading etc) and a Data Layer for driving Interactive/low-latency analysis(a traditional RDBMS like Redhsift/Vertica, or a materialized view layer like Apache Kylin or a OLAP index like Apache Druid).

Borrowing from the RADStack [3] paper a typical solution may look like the following:

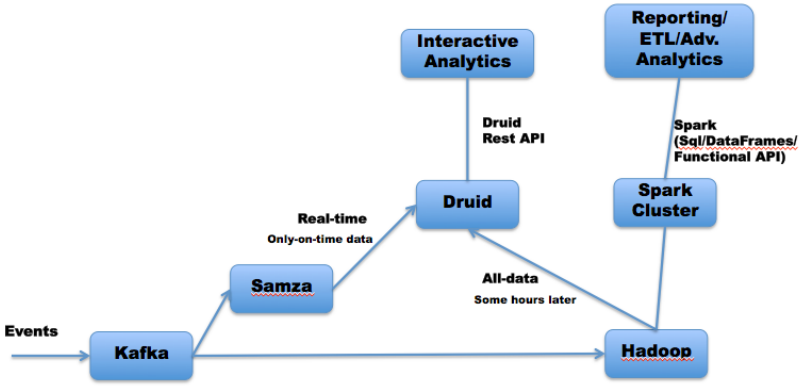


Figure 1: RADStack Architecture

The RADStack architecture leverages Druid for a separate serving layer for fast low latency Interactive Analytics. Whereas Spark's powerful programming

model offers much tighter integration between relational and procedural processing enabling users to express complex ETL, Reporting and Advanced Analytics(graph and machine learning) through a single interface.

**Is it possible to combine the two: provide Spark's rich programming model and Druid's acceleration?** In the rest of the Blog we layout an architecture that promises to do this. But first some details about Druid and Spark.

## Druid

Apache Druid [4] is data store designed for fast exploratory analytics on a very large wide data set(think event streams). Its key capabilities and underlying techniques are:

- a columnar storage format for partially nested data structures.
- an olap/multi-dimensional distributed indexing structure
- arbitrary exploration of billion-row tables with sub-second latencies
- realtime ingestion (ingested data is immediately available for querying)
- fault-tolerant distributed architecture that doesn't lose data.

Druid is architected as a group of services:

**Historical nodes** handle storage and querying on "historical" data (non-realtime).

**Realtime nodes** ingest data in real time. They are in charge of accepring incoming data and making it available immediately to Queries. Aged data is pushed to deep storage and picked up by Historical nodes.

**Coordinator nodes** monitor historical nodes and ensure that data is available, replicated and in a generally "optimal" configuration.

**Broker nodes** receive queries from clients and forward those queries to Realtime and Historical nodes. They merge these results before returning them to the client.

**Indexer nodes** form a cluster of workers to load batch and real-time data into the system

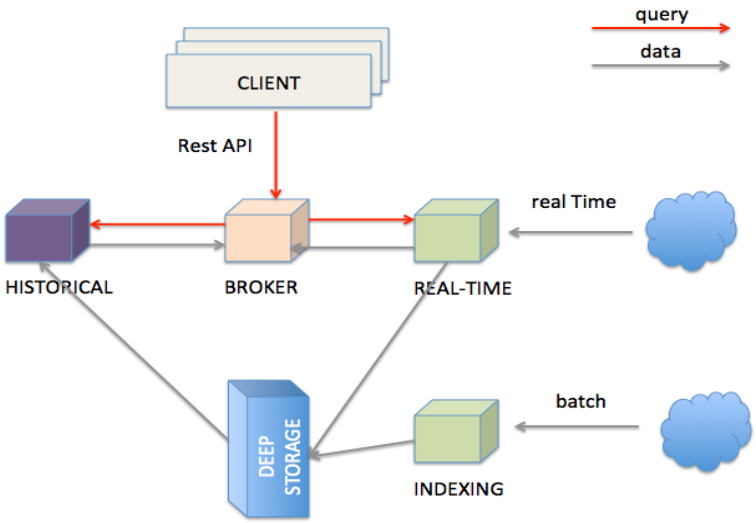


Figure 2: Druid Architecture

### Segments

Segments represent the fundamental storage unit in Druid. Data tables in Druid (called "data sources") are collections of timestamped events and partitioned into a set of segments, where each segment is typically 5–10 million rows. Druid partitions its data sources into well defined time intervals, typically an hour or a day, and may further partition on values from other columns to achieve the desired segment size. Segments are column oriented multi dimensional inverted indexes. As explained in the RADStack paper, for example for each publisher value a bitmap is maintained:

```
bieberfever.com -> rows [0, 1, 2] -> [1][1][1][0][0][0]
ultratrimfast.com -> rows [3, 4, 5] -> [0][0][0][1][1][1]
```

To know which rows contain bieberfever.com or ultratrimfast.com the two arrays are ORed together.

```
[1][1][1][0][0][0] OR [0][0][0][1][1][1] = [1][1][1][1][1][1]
```

Metrics are also stored in a column orientation:

```
Clicks -> [0, 0, 1, 0, 0, 1]
Price -> [0.65, 0.62. 0.45, 0.87, 0.99, 1.53]
```

Only the metric columns needed to answer the query are loaded. Also the entire metric column doesn't need to be scanned, only positions based on the dimensional predicates.

The combination of bitmap operations on dimensional predicates, columnar orientation and smart scans make answering slice-and-dice queries very fast: orders of magnitude faster than traditional row-oriented databases. This is how Druid is able to support interactive analysis workloads with sub-second performance.

## Spark

Apache Spark [5, 6] is an in-memory, general-purpose cluster computing system whose programming model is based on a LINQ [7] style API on datasets. It provides a very rich Datasets API in Java, Scala, Python and R, and a runtime engine that supports general execution graphs. On top of this core it provides several higher level components: Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.

Spark SQL [8] integrates relational processing with Spark's functional programming API. It provides a *DataFrame API* that can perform relational operations on both external data sources and built-in datasets. A second key component is a novel extensible optimizer called *Catalyst*. Catalyst makes it easy to add external data sources, optimization rules and data types for performing advanced analytics.

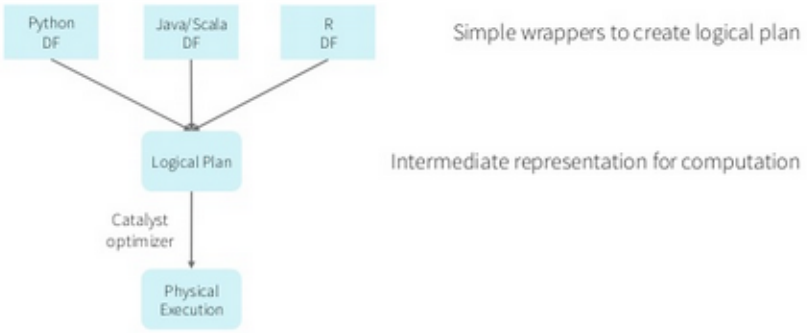


Figure 3: SPARK-SQL

### DataFrames

**DataFrames** are collections of structured records that can be manipulated using Spark's procedural API or using new relational operators which allow for rich optimizations. Other Spark components such as the machine learning library are being refactored to operate at the Dataframes API abstraction. Analysis expressed as DataFrame operations leverage several key benefits: automatic storage of data in columnar format that is significantly more compact than native java/python objects, logical and cost based optimizations provide by Catalyst, and code generation of expressions. In fact with Project Tungsten [9] the benefits of writing to the DataFrames API are only increasing: Spark is moving to an architecture where the DataFrames and Catalyst components will sit in between all higher level APIs and the runtime.

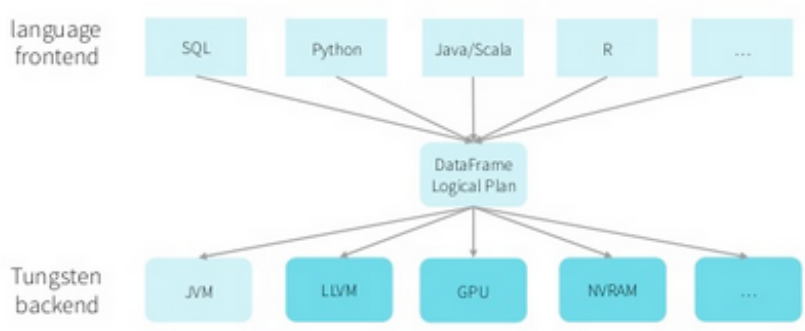


Figure 4: Project Tungsten

DataFrames support common relational operators, including projection (select), filter (where), join, and aggregations (groupBy). These operators all take expression objects that enable users to write expressions involving arithmetic, comparison, logical and user-defined operators. The ability to combine Relational operators with Scala, Java or Python code makes expressing your logic much more powerful and simpler than just using SQL.

### Catalyst

Catalyst's general tree transformation framework has four phases:

- plan analysis: entity and schema resolution.
- logical optimizations
- physical planning
- code generation to compile expressions to Java bytecode.

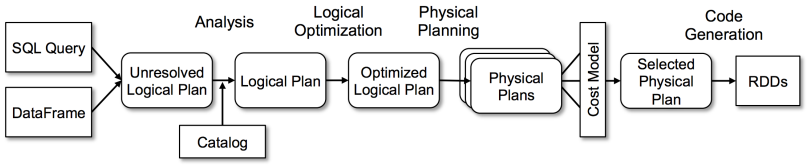


Figure 5: Catalyst: Phases of Query Planning

Central to Catalyst is its extensibility. Developers can add Rules for each phase of translation. The pattern matching and other functional aspects of Scala make writing new transformations rules quite easy. Catalyst goes further by providing two narrower extension mechanisms: Data Sources and User Defined Types that users can use without understanding all the details of Catalyst.

**DataSources** extend the Spark platform to work on *external* data. This could mean handling new data-formats (like CSV, Avro, Parquet) and also bridging to external data systems like an RDBMS via accesing data via JDBC. The interface let's user define different levels of capability of a DataSource. The basic

contract is for external data to be exposed as a DataFrame; with the ability for DataSource writers to support pushing Column Pruning and Predicates down to the source of the data.

## Spark and Druid: from RAD to Foundational

What we want is to **separate Query specification from Physical Plans**: specifically given a Logical Plan, where possible we want to rewrite the Plan to use the Druid Index. It shouldn't matter how the Plan was expressed: SQL, a custom DataFlow containing relational, machine learning operators etc. We setup a DataSource that wraps(and hence exposes the schema and data) of the **raw event** DataSet, but has access to the corresponding Druid Index. A companion Planning component than rewrites Plans on the **raw event** Dataset to utilize the Index where possible. This Datasource can handle all workloads: Interactive Queries, Reporting, and Advanced Analytics.

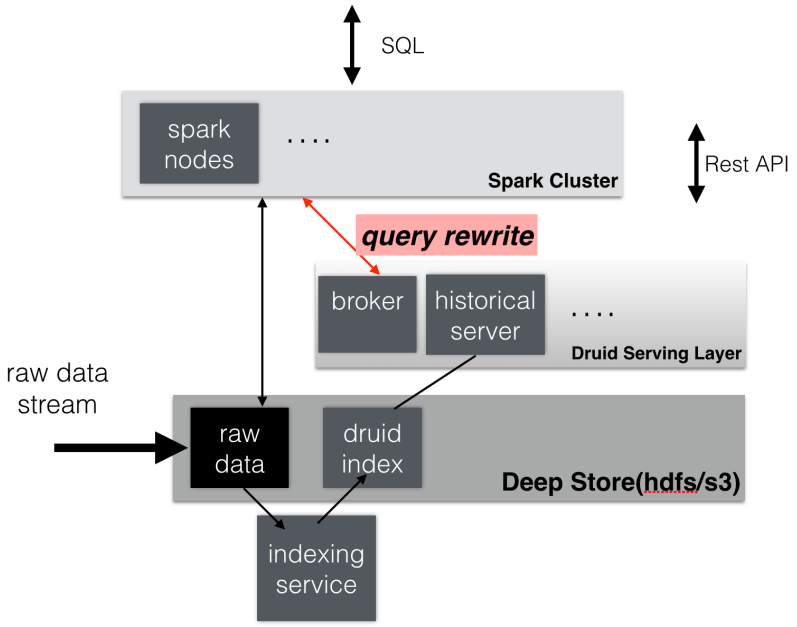


Figure 6: Spark Druid Overall Picture

Since this capability is available at the DataFrames level, it is available to all programming interfaces of the Stack: Spark-SQL, MLLib, GraphX and custom Applications/APIs. **This is the key contribution of this architecture: bringing the rich programming model of Spark together with the fast slice-and-dice capability of Druid.** The RADStack picture evolves to a state where the access method is decoupled from the physical capability of the underlying Data serving layer.

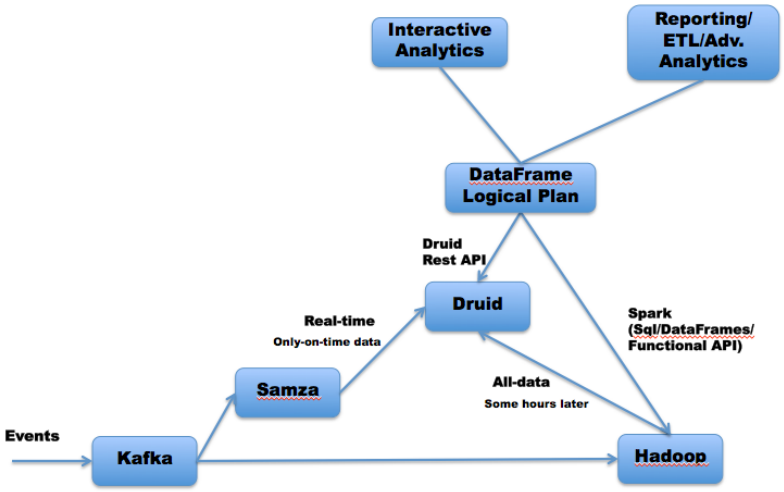


Figure 7: From RAD to Foundational

## Spark Druid Package

We have opened source the Spark Druid package [10] that has 2 components: (1) DruidDataSource is a Spark Datasource wraps the DataFrame that exposes the *raw* Dataset and also is provided with information about the Druid Index for this Dataset. (2) During planning, the DruidPlanner attempts to apply a set of rewrite rules to convert a Logical Plan on the raw dataset DataFrame into a DruidQuery.

Here is a example of defining a Druid DataSource:

Listing 1: Defining a Druid DataSource

```
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"
                      }     '
)
```

The raw dataset is exposed in the *orderLineItemPartSupplierBase* DataFrame. There is a Druid Index on this Dataset called **tpch**, the *l_shipdate* column is used as the time dimension for the index. It returns a DruidRelation a BaseRelation to the Spark engine The basic behavior of *DruidRelation* when asked for an RDD is to defer to the underlying DataFrame(orderLineItemPartSupplierBase in the above example). But if the DruidRelation has an attached DruidQuery, a query is run against the associated Druid Index and the returned result is

injected into the Spark pipeline as Spark Rows. During Planning the **Druid-Planner** attempts to convert an Aggregation Sub Plan into an equivalent Druid-Query. The details on how this happens the Rewrite Rules please refer to the detailed design document [11].

# A Deeper look at Query execution

But let's go over a specific Query so that you can see the impact of the Druid-DataSource and DruidPlanner. We have benchmarked a set of representative queries that contrast performance of queries being rewritten to use a DruidIndex vs. running the Queries directly against the **raw event** DataSet. The Benchmark is described in detail in a separate paper [12] on our website. We used the TPCH benchmark dataset, and converted the star-schema into a flattened(denormalized) transaction dataset. Consider a typical slice-and-dice query on this dataset:

```
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
```

This query involves dimensional predicates on `p_type`, `s_nation` and `c_nation`, it also has a time range of `1995-12-01 to (1997-12-01 - 90.days)`. It is looking at a set of metrics for each Supplier Nation in this region of the overall Sales Cube. This is quite typical query in a Interactive Analysis session: drilling into a particular region of the Cube(by applying a few dimensional predicates) and asking for a set of metrics for a particular set of Dimensional members.

The Logical Plan of this query is:

```
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,...
```

It involves Partition Pruning(since the raw dataset is partitioned by shipDate not all the partitions need to be read), a Filter, Project and Aggregate operation.

The rewritten Plan using the Druid Index looks like this:

```
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 benchmark shows that the rewritten Plan is an order of magnitude faster than the original execution Plan. It should be easy to understand this now:

bitmap operations on the inverted index structure quickly give the positions that must be aggregated. The columnar nature of the metrics make aggregation very fast. Finally the time partitioning of Segments means only the segments in the Query interval need to be processed.

# Conclusion

**Spark DataFrame** brings the separation of expressing Analytics from the Physical execution Plan: through the magic of **Catalyst** these Plans are sped up by techniques like: Columnar Storage, logical and cost based Plan rewrites, and code-generation. **Project Tungsten** only accelerates the advantages of this architecture: bringing even more runtime optimizations to bear. The **RADStack** Architecture shows how Interactive Analytics can be supported in a Big Data environment. By combining Spark and Druid, we bring the fast slice-and-dice capability of Druid to not just Interactive Analytics but to all Analytic workloads.

# References


[1] *Machine-generated data.*

[2] *50 Sensor Applications for a Smarter World.*

[3] *The RADStack: Open Source Lambda Architecture for Interactive Analytics.*

[4] Eric Tschetter Fangjin Yang. "A Real-time Analytical Data Store". In: *SIGMOD* (2014). URL: `http://static.druid.io/docs/druid.pdf`.

[5] Tathagata Das Ankur Dave Justin Ma Murphy McCauley Michael J. Franklin Scott Shenker Ion Stoica Matei Zaharia Mosharaf Chowdhury. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." In: *NSDI* (2012). URL: `http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf`.

[6] Michael J. Franklin Scott Shenker Ion Stoica Matei Zaharia Mosharaf Chowdhury. "Spark: Cluster Computing with Working Sets." In: *HotCloud* (2010). URL: `http://people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf`.

[7] M. Isard and Y. Yu. "Distributed data-parallel computing using a high-level programming language." In: *SIGMOD* (2009).

[8] Cheng Lian Yin Huai Davies Liu Joseph K. Bradley Xiangrui Meng Tomer Kaftan Michael J. Franklin† Ali Ghodsi Matei Zaharia Michael Armbrust Reynold S. Xin. "Spark SQL: Relational Data Processing in Spark". In: *SIGMOD* (2015). URL: `http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf`.

[9] *Project Tungsten: Bringing Spark Closer to Bare Metal.*

[10] *Spark Druid Package.* `https://github.com/SparklineData/spark-druid-olap`.

[11] *Spark Druid: a suitable match.* `https://github.com/SparklineData/spark-druid-olap/blob/master/docs/SparkDruid.pdf`.

[12] *Spark Druid Benchmark*. https://github.com/SparklineData/spark-druid-olap/blob/master/docs/benchmark/BenchMarkDetails.pdf.