Spark and Druid: a suitable match

Harish Butani

August 28, 2015

Druid [1] was designed to handle the ingest and analysis of large amounts of **Event** data. This is a key capability in the AdTech/Marketing and Internet-of-Things(IOT) spaces. In the AdTech arena events generated from human activity are analyzed for doing User targeting, Campaign attribution, Site optimization etc. In the IOT arena event streams and analysis are at the heart of myriad of Applications for Smart Cities, Smart Environment, Smart Water etc.

Event Datasets are like flattened(wide denormalized tables) multi-dimensional Cubes. Druid provides fast slice and dice capability on the raw event data by building a multi-dimensional OLAP index that is primarily partitioned on Time. OLAP indexes are a technique that is used to speed-up slice and dice queries(MDX) on traditional cubes. SAP BW Accelerator TREX Engine [2](the precursor to SAP Hana) is an example of another OLAP indexing technology. Druid provides a REST based Query interface over its Index. This is similar to SQL and there have been efforts to provide SQL-like [3] interfaces on top of Druid; but a full fledged SQL interface is missing.

On the other hand Apache Spark [4] has a very rich LINQ [5] like programming model for expressing data analysis. It has rich SQL capabilities that closely matches Hive SQL: standard join support, datatypes, analytical functions, Cubing/Rollup, jdbc/odbc access etc. The DataSources API [6] enables federating external Datasets; enabling bridging data from disparate systems and applying the analytic capabilities of Spark on all of the data.

There are 2 scenarios where we see the combination of Druid and Spark adding a lot of value:

- For existing deployments of Druid, expose the Druid Index as a DataSource in Spark. This provides a proper SQL interface(with jdbc/odbc access) to the Druid dataset; it also enables analytics in Spark(SQL + advanced analytics) to utilize the fast navigation/aggregation capabilities of the Druid Index. The second point is the key: the goal is **fast SQL++ access on a denormalized/flat event Dataset**. One that marries the rich programming model of Spark with the fast access and aggregation capabilities of Druid. *This paper focuses on this scenario. Our initial development is for this scenario.*

- Longer term , we envision using the Druid technology as an OLAP index for Star Schemas residing in Spark. This is a classic OLAP space/performance tradeoff: a specialized secondary index to speed up queries.

The rest of the paper describes the details of the **Spark Druid Datasource** and a set of **Plan Rewrite Rules**. We demonstrate how Spark-SQL Queries written against the **raw event** Datasets are rewritten to utilize the Druid Index. We benchmarked a set of representative Queries on a 45GB dataset, we share are findings in the Benchmark section.

The components in this paper are available in our Spark-Druid package [7] which is open source, and which we plan to release as a Spark Package shortly.

Brief Introduction to Spark, Druid

Druid

Apache Druid 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:

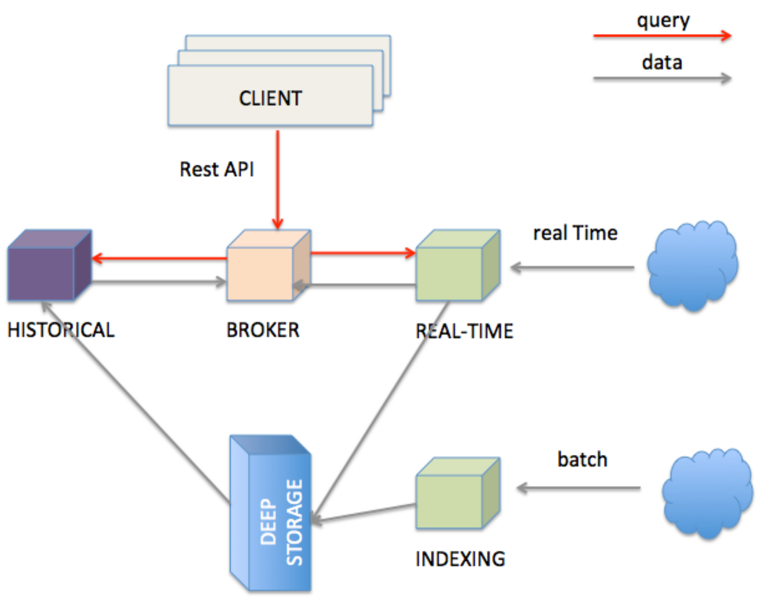
Historical nodes handle storage and querying on "historical" data (non-realtime).

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.

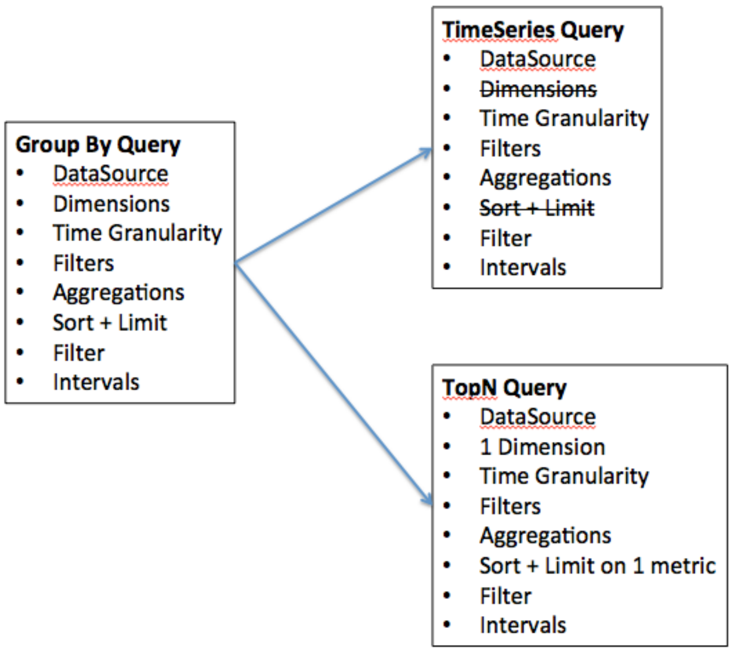
Broker nodes receive queries from clients and forward those queries to Realtime and Historical nodes. They merge these results before returning them to the client.

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



Druid Query Model

Queries are made using an HTTP REST style request. **Group By** queries are the most generic type of queries, use them to specify the DataSource, the Dimensions to Group By, the Time Granularity, Filters on Dimensions, Aggregations, how to order and limit results, and the time intervals for which this query needs to run. Here are examples of TPCH queries expressed as Druid Queries. **Timeseries** and **TopN** are Druid Query types that can be used when certain constraints are met; they provide better performance than running the Group-By Query. Timeseries are significantly faster than groupBy queries for aggregations that don't require grouping over dimensions. For grouping and sorting over a single dimension, topN queries are much more optimized than groupBys.



We provide a Query Data model in Scala to define queries, and the mechanics to interact with Druid to execute these Query Specifications and fetch results.

Spark

Apache Spark is an in-memory, general-purpose cluster computing system whose programming model is based on a LINQ 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.

The DataSources API enables Spark to integrate natively with a large number of external sources. DataSources have been written for Cassandra, JDBC DS, CSV etc. The DataSources abstraction provides a mechanism by which processing in the form of *predicates and column pruning* can be pushed down to the external system where the Data resides.

Spark and Druid

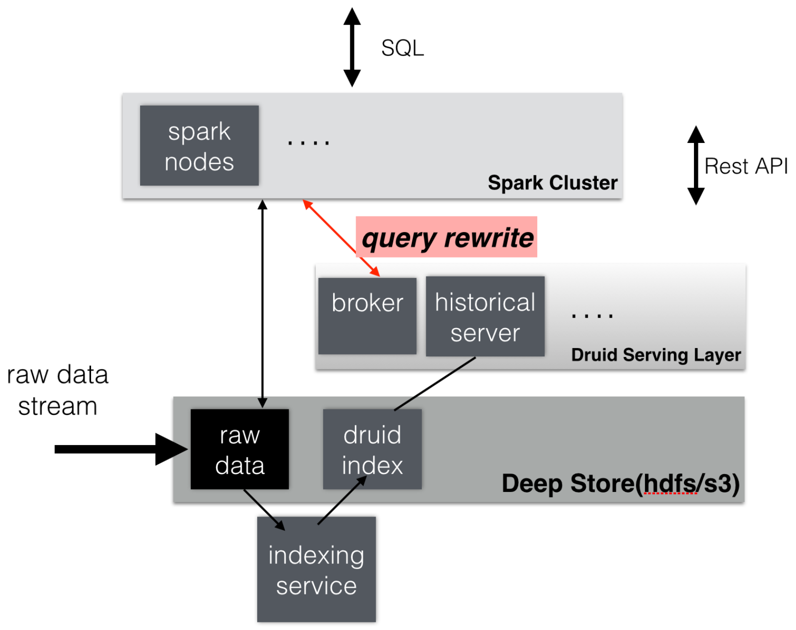
An obvious starting point is to just expose a Druid Index as a Spark DataSource. This seems like a useful thing to do: it enables proper SQL access; deeper analytics on the Event data is enabled without having to copyof the event data(and more painful manage the copy) . But this is not a very useful solution for the following reasons:

- the DataSource mechanics only allow predicate pushdown and column pruning; so aggregations have to be done in Spark; **one of the big strengths of the Druid index is nullified.**

- **This treats Druid as the primary source of the data. In fact in most cases this is not the case.** The usual Data setup is for raw data to land in hdfs or s3, for data to be indexed and possibly aggregated to a higher time grain. For example a Druid index may have aggregated information up to an hour or day granularity.

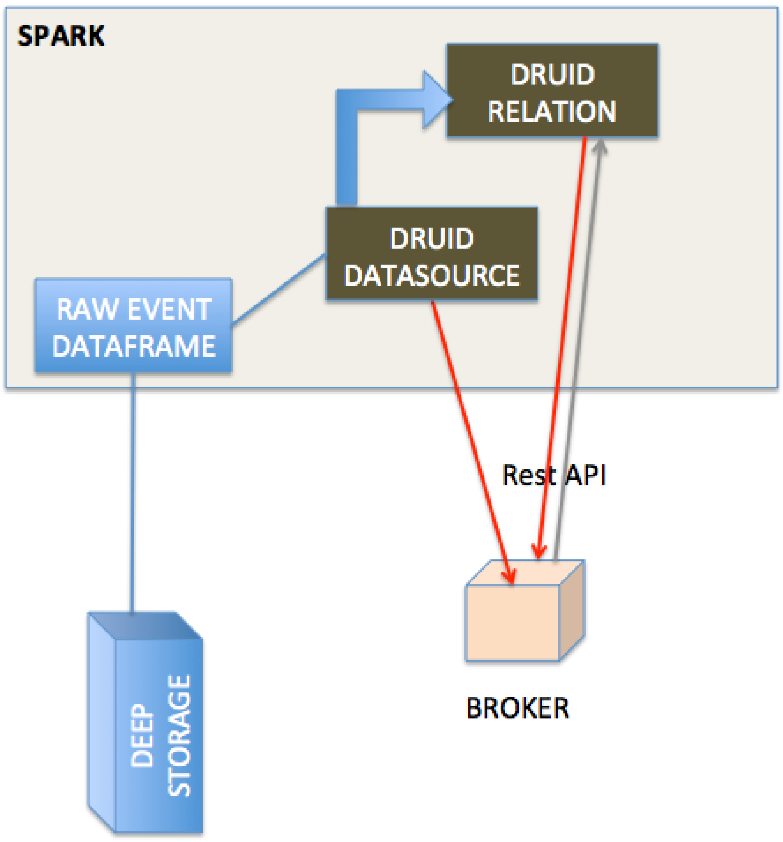
The fundamental problem with the Datasource only approach is that it doesn't treat Druid as an Index. What we want is to **make it appear that the raw event DataSet is being accessed, and where possible to rewrite Query Plans on this DataSet to use the Druid Index**. The overall picture is:

- **raw event** data is landing in hdfs/s3, and a Druid Index is kept upto date.
- the Event data is exposed in the Spark Analytical platform as residing on the Deep Storage layer: hdfs/s3.
- 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.



Druid DataSource for Spark

DruidDataSource is a Spark Datasource that enables users to utilize the Druid Index to accelerate OLAP style queries on the underlying **raw event** Dataset. It wraps the DataFrame that exposes the *raw* Dataset and also is provided with information about the Druid Index for this Dataset.



The DataSource is configured with the following parameters:

Name	Description
sourceDataFrame	The DataFrame that represents the raw Data
druidHost/Port	Information on how to connect to the Druid Broker
druidDatasource	Name of the Druid Index for the raw dataset
timeDimensionColumn	The column from the raw dataset that is the time dimension in the Druid Index
columnMappping	a Map for mapping raw dataset column names to column names in Druid.

Other parameters are also available/will be added to configure rewrites and Druid behavior like functionalDependencies, maxCardinalityPerQuery, maxResultCardinality etc. These will be documented in the future.

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.

When Spark asks the **Druid DataSource** to create the Relation: it connects to Druid, reads the metadata about the specified Druid datasource and sets up a DruidRelationInfo metadata object. 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 it has an associated DruidQuery, it returns a DruidRDD. A DruidQuery encapsulates a Druid Query specification, along with a List of intervals on which to apply the Query, and information on how to map the result into Spark Rows. DruidRDD is the bridge between Spark and Druid. It runs the DruidQuery on Druid for each interval(DruidRDD returns the results of each interval in a separate partition). For each Partition the compute call invokes the Druid Broker with the Druid Query, the results are converted into a Iterator of Spark Rows.

During planning, the DruidPlanner applies a set of rewrite rules to convert a Logical Plan on the raw dataset DataFrame into a DruidQuery.

Query Rewrites

Spark SQLContext allows the Spark Planner to be configured with extra physical plan generation rules. These are applied before built-in Physical transformation. We add the DruidStrategy to the SparkPlanner.

The DruidStrategy

This relies on the DruidPlanner to possibly convert a LogicalPlan into a DruidQueryBuilder. If a LogicalPlan has an equivalent DruidQueryBuilder, than this is converted into a SparkPlan with the following steps:

1. Setup a DruidQuery object: this contains the QuerySpec (a scala data structure that matches the Druid json information model for expressing queries), and the intervals this Query needs to run on.

2. Setup a DruidRelation with the the DruidRelationInfo metadata object and DruidQuery object.

3. Setup a Physical Plan that looks like

```
Project
  PhysicalRDD(druidRelation.buildScan)
```

The PhysicalRDD wraps the RDD provided by the DruidRelation. The Projection takes care of any dataType mappings and evaluating expressions on aggregation from the Aggregation Operator original Plan..

DruidPlanner

The DruidPlanner is the the entry point for the Druid rewrite functionality. It is a container of DruidTransforms. In order to enable rewrites the user needs to invoke `DruidPlanner(sqlContext)`. This registers DruidStrategy with the SparkPlanner. A DruidTransform is responsible for converting a Logical Plan into a DruidQueryBuilder. A DruidQueryBuilder is a case class that captures information about a Druid Query. It also captures mapping information from Spark Expressions to Results coming out of Druid: including dataType and column name mappings. There are several DruidTransforms to convert different Plan trees to a DruidQuery, but the Logical Plan must at least contain an Aggregation Operator. More on this in the Query Rewrite Rules section.

Mapping Druid results into Spark Rows

Query Building: Column Name, Type mapping

The DruidQueryBuilder mainatins a map from the Druid Query Result columnName to the triple: (Expression, spark DataType, druid DataType):

- Expression is the Catalyst Expression from the original Plan that the Druid column in the Result row represents.
- The DataType of the Expression in the original SQL plan.
- The DataType of the value returned by Druid.

The 2 datatypes need not match; during rewrite a check is made to see if the conversion from the Druid datatype to Spark Expression datatype is valid. If not, the rewrite doesn't happen. This map is populated as expressions from the Aggregate Operator are added to the DruidQueryBuilder.

Setting up the Output Schema of the PhysicalRDD Operator that wraps the Druid RDD

The schema for the PhysicalRDD Operator is formed by creating a StructType from each of the columns in the output Map maintained by the DruidQueryBuilder. For Grouping Expressions that were AttributeReferences in the original Plan, we reuse their ExprIds; for non AttributeReferences new ExprIds are generated. This way any resolved AttributeReferences above the replaced Plan SubTree are still valid and point to the correct child Attribute in the rewritten Plan.

Projection on top of the PhysicalRDD Operator.

A Projection Operator is added above the PhysicalRDD Operator to:

- provide the same schema as the original Aggregate Operator. (or the Ordering/Filter Operator above the Agg.Op in case of having/order/limit rewrites)

- To ensure Attribute names, ExprId and DataTypes match what was in the original Operator.

The ProjectionList is formed from the aggregation expressions of the original Agg. Operator. Any expressions that were mapped to Druid Result columns are replaced by AttributeReferences to the child PhysicalRDD Attributes. The following rules are followed:

- If needed the AttributeReference is wrapped in a cast to convert to the original Spark Plan's dataType.
- AttributeReferences in the original Plan carry the original ExprId, so that references above this Operator remain valid. Names from the original AttributeReference are also maintained by wrapping the new AttributeReference in an Alias.

TPCH Flattened Cube example

We explain the Rewrite rules by giving examples from the following setup. Consider a raw transaction log that is based on the TPCH benchmark specification

Listing 2: The TPCH denormalized DataFrame

```
CREATE TEMPORARY TABLE orderLineItemPartSupplierBase(
    o_orderkey integer, o_custkey integer,
    o_orderstatus string, o_totalprice double,
    o_orderdate string, o_orderpriority string,
    o_clerk string, o_shippriority integer,
    o_comment string, l_partkey integer,
    l_suppkey integer, l_linenumber integer,
    l_quantity double, l_extendedprice double,
    l_discount double, l_tax double,
    l_returnflag string,l_linestatus string,
    l_shipdate string, l_commitdate string,
    l_receiptdate string, l_shipinstruct string,
    l__shipmode string, l_comment string,
    order_year string, ps_partkey integer,
    ps_suppkey integer,ps_availqty integer,
    ps_supplycost double, ps_comment string,
    s_name string, s_address string,
    s_phone string, s_acctbal double,
    s_comment string, s_nation string,
    s_region string, p_name string,
    p_mfgr string, p_brand string,
    p_type string, p_size integer,
    p_container string, p_retailprice double,
    p_comment string, c_name string ,
    c_address string , c_phone string ,
    c_acctbal double ,c_mktsegment string ,
    c_comment string , c_nation string ,
    c_region string)
USING com.databricks.spark.csv
OPTIONS (
  path "tpchFlattenedData_10/orderLineItemPartSupplierCustomer",
  header "false", delimiter "|"
)
```

This is a single transaction table that is formed by denormalizing(flattening) the TPCH Star Schema. We have a TpchGen tool for creating a flattened transaction table from an existing Tpch Star schema.

Also assume there is a Druid Index built for this DataSet and is exposed in Spark as a DruidDataSource

Listing 3: TPCH 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"
                     }     '
)
```

So queries are rewritten against the 'orderLineItemPartSupplier' table. For example TPCH Q1 is written as:

Listing 4: Sample Query

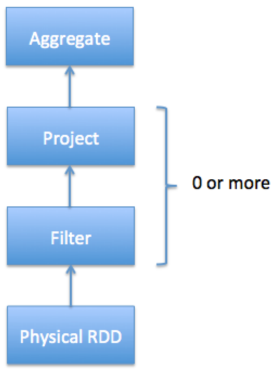
```

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
```

Without the DruidPlanner configured these queries will run as if they are issued against the underlying sourceDataFrame, in this case against the wrapped DataSource 'orderLineItemPartSupplierBase'.

Query Rewrite and Validation Rules

Plans that can be rewritten must have the following core structure.



The base of the Plan must be a Physical RDD Operator on a DruidRelation, followed by 0 or more Project/Filter criteria, followed by an Aggregation. Only plans with this core structure are considered for rewrite. On top of the Aggregation, there can optionally be a Filter(representing the SQL having clause), a Sort and a Limit.

Validation 1: Base table column validation

Columns referenced in the Project below the Aggregate must have a corresponding column in the Druid Index.

Rewrite 2: Filter Rewrite

The Filter predicates are combined into **Conjunctive Normal Form**. An attempt is made to rewrite each conjunct. If any conjunct cannot be rewritten, then the Plan is not rewritten.

Rewrite 2.1: Interval condition rewrite A predicate of the form `compOp(dateTime(timeDim), literalDateTime)` is extracted as an *time Interval* of the Druid Query.

Where 'compOp' can be the following functions: `dateIsBeforeFn, dateIsBeforeOrEqualFn, dateIsAfterFn, dateIsAfterOrEqualFn`. The comparison needs to be on the column that is the time dimension in the Druid Index(in our example the 'l_shipDate' column). The literal-date is an expression representing a date. It can be a literal date specified with `dateTime, dateTimeWithTZFn, dateTimeWithFormatFn, dateTimeWithFormatAndTZFn` optionally followed by(+/-) a *Period* specification. For example the following predicate is translated to the Interval ("1992-12-01", "1997-09-02") :

```
dateIsBeforeOrEqual(
   dateTime(‘l_shipdate‘),
   dateMinus(
     dateTime("1997-12-01"),
     period("P90D")
   )
)
```

It is much easier to read when specified using spark-dateTime dsl

```
dateTime('l_shipdate) <= (dateTime("1997-12-01") - 90.day)
```

Currently we only translate the SQL predicates into a single interval. The QueryIntervals class is setup to handle multiple intervals. In the future we plan to handle a disjunction of date Predicates in each conjunct.

Rewrite 2.2: Dimension Filter rewrite Predicates of the form `dimCol compOp Literal` or `Literal compOp dimCol` are converted into Filter Specifications on the Druid Query. The column being compared must be a dimension column in the Druid Index. The comparator operator needs to be `<,>, <=, >=,=`. Comparison predicates can be combined with logical `and, or` operators.

Rewrite 3: Grouping Expressions

A Group-By expression can be on a Druid index dimension or a dateTime expression on a regular or time dimension in the Druid index. The dateTime expression must be of the form `dateElem(dateTimeFn(col))`. The 'dateTimeFn' form must be `dateTime, withZone(dateTime...`, that is a dateTime expression or a dateTime with Timezone application. The column must be a dimension or the time column of the Druid Index. The element being extracted can be any of:

```
era, century, yearOfEra, yearOfCentury, year, weekyear,
monthOfYear, monthOfYearName,weekOfWeekyear,
dayOfYear, dayOfMonth, dayOfWeek, dayOfWeekName,
hourOfDay, secondOfMinute
```

An expression on a dimension is expressed as a DefaultDimension Specification the DruidQuery. While a time element expression is converted to a TimeFormatExtraction Specification.

Rewrite 4: Aggregation Expressions

From the aggregation list we extract the AggregateFunction invocations, and attempt to translate them to Druid Aggregation and PostAggregation Specifications on the Druid Query. On the translated Plan a Project operator is placed on top of the Druid Relation to compute any expressions that the Aggregate Function invocations were part of. So for the expression `sum(p_retailprice) - 5`: the `sum(p_retailprice)` is pushed to the Druid Query; the subtraction on the sum is handled in the Project Operator on top.

The following rules are used to translate Aggregate functions

Count `Count(1)` is translated to a Cardinality Aggregation Specification.

Sum, Min, Max The aggregation must be on a Druid Metric column. The dataType of the expression must be convertible from the Druid metric dataType without loss of precision. The expression is translated to a Function Aggregation Specification on the Druid Query.

Avg This has the same constraints as Sum/Min/Max. It is converted to a Post Aggregation Specification of dividing the Sum by the Count.

CountDistinct Is converted to Cardinality Aggregation Specification. Druid uses HyperLogLog to estimate this. So in the future we will add a parameter to the DataSource, so users can control if this rewrite should be allowed.

Rewrite 5: Having predicates (TBD)

Predicates on the Aggregation expressions will be pushed down as Having Specifications in the Druid Query.

Rewrite 6: Sort Operator (TBD)

A Sort Operator on top of Aggregation will be pushed down as a Limit Specification in the Druid Query.

Rewrite 7: Limit Operator (TBD)

A Limit Operator on top of a Sort will be pushed down as a limit value on the Limit Specification in the Druid Query.

Rewrite 8: Enhanced Time Granularity and Interval Handling

We currently assume that the Druid Index has the same Time Granularity and Range as the **raw** data. This is obviously not necessary, and in practice an uncommon way to setup the Index. More likely, the Index is on a Grain(hourly, daily) higher than the raw events. Also index for old data maybe removed for space reasons.

Shorter Time Range for Druid Index

It is likely that the DruidIndex is maintained for a smaller Time window like the last year; whereas the raw dataset is for much longer time window. In such cases the original Plan should be converted into a **union all Plan**. The component queries being a Druid Query on the Time Window that is in the Druid Index(and intersects with the Query predicate) and a Spark Query on the raw event DataSource for the remaining Time Window.

Druid Index on a higher Time Grain.

It is likely that the Druid Index doesn't hold raw data, but is aggregated up to a minimum time grain such as an hour or a day. In this the original Plan can only be rewritten if the Query has a Time Aggregation that is at a higher grain than the granularity in the Druid Index.

Benchmark

We ran a benchmark on 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. We picked a mixture of aggregation Queries: slice-and-dice queries common in OLAP scenarios, and aggregation queries common in Reporting workloads. Our expectation was for Druid to show significant benefit for the former queries, and Spark to win out on latter type of queries.

We have attempted to make this test as fair as possible: **by not taking any advantages of preaggregations and column pruning in Druid, and by using in memory caching when the queries run only in Spark**. We ran the 2 scenarios on the same cluster: for the Druid run we gave the cluster node resources to Druid History servers, for the Spark run we ran Spark Executors on the worker nodes.

For the benchmark we used the TPCH benchmark dataset, datascale 10G. We converted the 10G star schema into a flattened(denormalized) transaction dataset. For Spark we further processed the data to setup a Partitioned table, stored in Parquet format; the table is partitioned by month. The Druid Index was created using the HadoopDruidIndexer.

The dataset sizes are:

TPCH Flat TSV	46.80GB
Druid Index in HDFS	17.04GB
TPCH Flat Parquet	11.38GB
TPCH Flat Parquet Partition by Month	11.56GB

The Benchmark is described in detail in a separate paper [8] on our website. Here we summarize the Queries, and present our findings.

Queries

We chose the queries to test the performance differences for *time intervals*, *dimension filters*, *time slice aggregation*, and *dimensional aggregations*. We have tested for TPCH queries Q1, Q3, Q5, Q7 and Q8. These queries have been altered so that they can be rewritten as Druid queries. As explained below, these don't change overall benchmark conclusion, if anything the numbers will skew further in favor of Druid once we start to push Sorts and Limits to Druid.

Alterations made to queries:

- Formulas in aggregations have been removed. For e.g. `sum(l_extendedprice*(1-l_discount))` is simply written as `sum(extendedprice)`. This is because we haven't implemented the rewrite to handle aggregation formulas. This has a minor impact on query performance; so the overall benchmark analysis will not change.
- Since we haven't implemented the rewrites, we have removed **order by, limit** clauses from the queries. Again this shouldn't change the overall benchmark conclusion; in fact once we can push down order and limit to Druid, the rewritten query times will reduce.

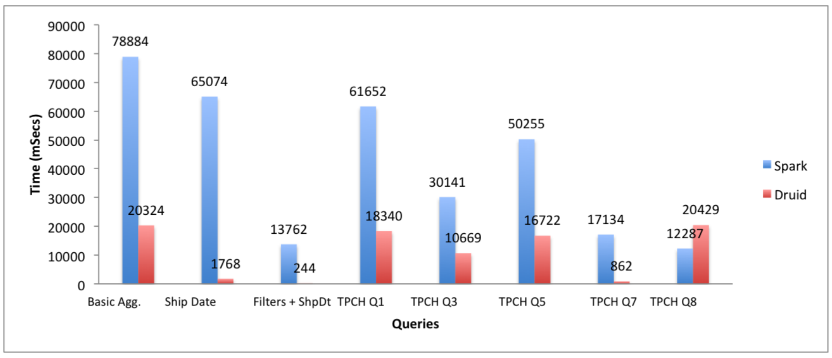
We have summarized the Query forms below. `Appendix A: Query Details` describes the Queries in detail.

Query Summary:

Query	Interval	Filters	Group By	Aggregations
Basic Aggregation.	None	None	ReturnFlag LineStatus	Count(*) Sum(exdPrice) Avg(avlQty)
Ship Date Range	1995-12/1997-09	None	ReturnFlag LineStatus	Count(*)
SubQry nation, pType ShpDt Range	1995-12/1997-09	P_Type S_Nation + C_Nation	S_Nation	Count(*) Sum(exdPrice) Max(sCost) Avg(avlQty) Count(Distinct oKey)
TPCH Q1	None	None	ReturnFlag LineStatus	Count(*) Sum(exdPrice) Max(sCost) Avg(avlQty) Count(Distinct oKey)
TPCH Q3	1995-03-15-	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(exdPrice)
TPCH Q8	None	Region Type O_Date	ODate.Year	Sum(exdPrice)

The Queries are described in detail in the `Query Details` section.

Benchmark Results



Observations about the performance:

- The *SubQuery + filters + ShpDt Range* query gives the most benefit when rewritten to Druid. This is not surprising, as this Query is tailor-made for Druid. The **Interval** and **Dimension** predicates fully leverage the Segment pruning and inverted index layout of Druid.

- The *Ship Date Range* query also shows a significant boost when run on Druid. This is surprising since Spark partition pruning should have nullified the segment pruning Druid will do. We ensure that Spark is doing partition pruning, by explicitly adding *shipYear* predicates to Spark. Is this attributable to aggregations being quicker in Druid, or is this the overhead of predicate evaluation that Spark has to do for each row? Another aspect maybe that we didn't enable the Code Generation option for Catalyst expressions, doing so may bridge this gap.

- *TPCH Q7* shows a significant performance boost when running on Druid: milliSeconds vs. 10s of seconds. The difference is in applying the filter to each row when running on Spark vs. using the inverted index to quickly find what values to aggregate. The **year** level aggregation should have the same optimizations in play: Map side aggregation in the case of Spark, should significantly reduce the amount of rows being shuffled; and in the case of Druid the amount of data shipped to the Broker is small, because only year level aggregations are being shipped. **So queries with just dimensional predicates, and small output cardinality also get a significant boost when rewritten as Druid queries**.

- For *TPCH Q3*, *TPCH Q5* and *TPCH Q8* there is an improvement, but nowhere close to the boost for **Q7**. **This is because the OrderDate predicate is translated to a JavascriptFilter**. We need to look into ways of translating such predicates to a native java function.

- Queries *Basic Agg.* and *TPCH Q1* show some improvement. The Count-Distinct is translated to a Cardinality Aggregator which is an approximate count. This is definitely an advantage for Druid, for large cardinality dimensions. We need to test against Spark using the HyperLogLog aggregator.

We have tried to make the comparisons as fair as possible: by indexing at the lowest grain, and making the Druid index contain all columns. We also partitioned and cached the Spark DataFrame. Inspite of these steps, we find that a **Basic Aggregation** computation(even with the Count-Distinct removed) is faster in Druid. Part of the issue is that Spark needs more memory and compute resources then Druid; several Query runs had failures, causing Executors to crash and hence tasks to be redone. But the differences are not all explained by this, even if we consider minimum times from the Spark runs, Druid still performs better. They could be explained by the fact that we didn't run with Code Generation enabled. We plan to run a followup test with Code Generation turned on. We expect that for *Basic Agg.* and *TPCH Q1* Spark to perform as well as Druid. But for Queries involving **Dimensional** and **Interval** slices, Druid is still an order of magnitude faster(a seconds vs. 10 seconds). Building native functions in Druid will expand the queries that gain this kind of performance boost. Queries that have **Interval** and **Dimensional** predicates are very common in OLAP, as analysts browse the Cube applying filters, Drilling Down and Up.

Conclusion

Future work

References

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

[2] *SAP Business Warehouse Accelerator*. `http://scn.sap.com/community/bw-accelerator`.

[3] *SQL4D*. `https://github.com/srikalyc/Sql4D`.

[4] *Spark, Lightning-fast cluster computing*. `http://spark.apache.org/`.

[5] *LINQ (Language-Integrated Query)*. `https://msdn.microsoft.com/en-us/library/bb397926.aspx`.

[6] *Spark SQL Data Sources API Unified Data Access for the Spark Platform*. `https://databricks.com/blog/2015/01/09/spark-sql-data-sources-api-unified-data-access-for-the-spark-platform.html`.

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

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