Metamarkets Druid

By: Jay Wang

I. Background

Metamarkets is a tech company based in San Francisco, CA that provides advertisement analytics to its customers. Their web-based analytics console can quickly perform drill-downs and roll-ups of high dimensional data sets. Druid is the data store that powers this console. From their website, "Druid is the distributed, in-memory OLAP data store" Metamarkets built to serve their ad analytics needs.

NOTE *- Roll-up and Drill-down are two types of OLAP operations. Roll-up means aggregate/summarize the data along a dimension; one rolls-up on an attribute such as sales or location. Drill-down/up allows the user to navigate among levels of data from the most summarized (up) to the most detailed (down)* [1]*. For example, if in the data the time dimension is broken down into months, one can split (drill-down) the monthly sales totals into daily sales totals* [2]*.*

Before Metamarkets built Druid, the company tried two other routes – RDMS and NoSQL. Option One (RDMS) failed because when data was not in cache, a query couldn't be mapped to a pre-aggregated table, which resulted in slow performance. Option Two (NoSQL) failed because high dimensional online data processing requires pre-computing an exponentially large amount of data.

Ultimately, Metamarkets reached the bane of database systems – "*Keeping everything in memory provides fast scans, but it does introduce a new problem: machine memory is limited*." Thus, the evident solution was to distribute data over multiple machines. This crafted the initial requirements for Druid: (1) ability to load up, store, and query data sets in memory, (2) parallelized architecture to allow more machines, (3) parallelized queries to speed full scan processing, and (4) no dimensional tables to manage [3]. Interestingly, when Druid was first built, it was all in-memory all the time, yet the price-performance tradeoff grew out of control. Metamarkets then added the ability to memory map data and allow the OS to handle paging data in and out of memory on demand [4].

---

[1] http://en.wikipedia.org/wiki/OLAP_cube#Operations
[2] http://inf.ucv.ro/~fgorunescu/courses/DM/t5.pdf
[3] http://metamarkets.com/2011/druid-part-i-real-time-analytics-at-a-billion-rows-per-second/
[4] https://github.com/metamx/druid/wiki/Design

II. Architecture

One of they keys to Druid's speed is how it represents its data. There are two types of data – *alpha* represents raw, un-aggregated event logs, *beta* is the partially aggregated derivative. Because the *beta* is much more potent and compressed, Druid keeps it in memory.

alpha Example [5]:

timestamp	publisher	advertiser	gender	country	.. dimensions ..	click	price
2011-01-01T01:01:35Z	bieberfever.com	google.com	Male	USA		0	0.65
2011-01-01T01:03:63Z	bieberfever.com	google.com	Male	USA		0	0.62
2011-01-01T01:04:51Z	bieberfever.com	google.com	Male	USA		1	0.45
...							
2011-01-01T01:00:00Z	ultratrimfast.com	google.com	Female	UK		0	0.87
2011-01-01T02:00:00Z	ultratrimfast.com	google.com	Female	UK		0	0.99
2011-01-01T02:00:00Z	ultratrimfast.com	google.com	Female	UK		1	1.53
...							

beta Example [2]:

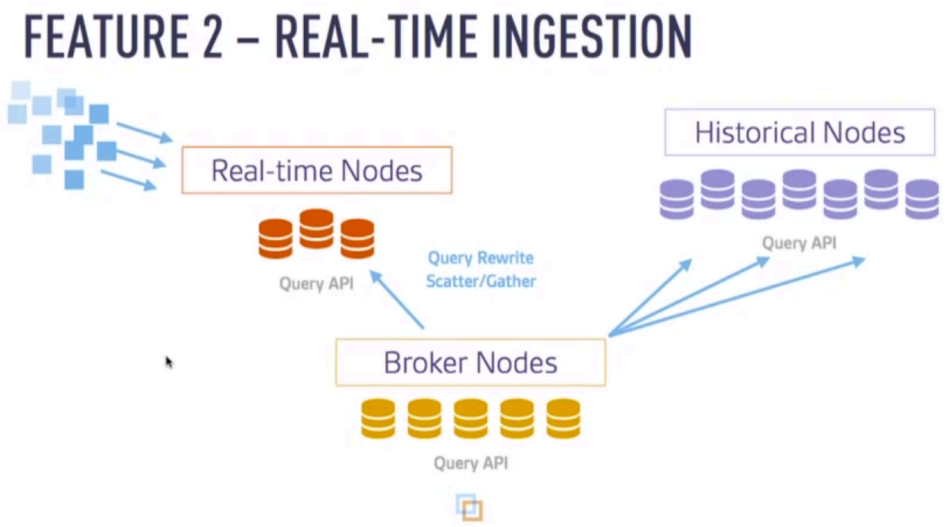
timestamp	publisher	advertiser	gender	country	impressions	clicks	revenue
2011-01-01T01:00:00Z	ultratrimfast.com	google.com	Male	USA	1800	25	15.70
2011-01-01T01:00:00Z	bieberfever.com	google.com	Male	USA	2912	42	29.18
2011-01-01T02:00:00Z	ultratrimfast.com	google.com	Male	UK	1953	17	17.31
2011-01-01T02:00:00Z	bieberfever.com	google.com	Male	UK	3194	170	34.01

Druid also utilizes bitmap indicing to support drill-downs on specific dimensions. It maintains a set of inverted indices that allows for quick calculation (using AND & OR operators). The inverted index enables Druid to scan a limited subset of rows – scans that are also distributed. Moreover, Druid's performance requires a lot of memory, which is achieved through dynamically distributing data across a cluster of nodes. As the data set grows, we can horizontally expand by adding more machines. To facilitate rebalancing, Druid takes chunks of *beta* data and indexes them into segments based on time ranges. Metadata about segments is stored in the query layer. All these segments are persisted into the S3 storage system that is accessible from all nodes. Moreover, if a node goes down, Zookeeper coordinates the live nodes to reconstitute the missing *beta* set [2].

Realtime Ingestion is done in Druid through the Historical and Realtime nodes. Both nodes have data for a time period they are responsible for. For example, the time segment can be one hour. In this case, a Realtime node would have data for one hour and that hour includes right now. On some time bound, the Realtime nodes will push their data segments over to the

[5] http://metamarkets.com/2011/druid-part-deux-three-principles-for-fast-distributed-olap/

historical nodes. Ultimately, the cluster combines both Realtime and Historical nodes. Here is a diagram [6]:



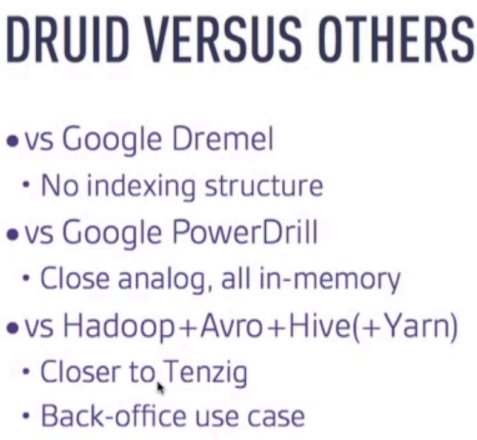
Druid has five types of nodes – compute, realtime, master, broker, and indexer. **Compute** nodes are the workhorses that handle storage and querying on historical data. **Realtime** nodes ingest data in real-time and focus on making the stream of incoming data immediately queryable by the system. **Master** nodes serve as the main coordinators. They look over groupings of computes and make sure that data is available, replicated, and in an "optimal" configuration. **Broker** nodes understand the data layout across all of the other nodes in the cluster and re-write and route queries accordingly. **Indexer** nodes form a cluster of workers to load batch and real-time data into the system as well as allow for alterations to the data stored in the system. As of Nov 2012, these nodes are still a work in progress. Realtime and HadoopDruidIndexer are the two entities handling indexing right now [7].

III. Use Cases

Druid is a system built to allow fast access to large sets of seldom-changing data. The service is designed to be always running. Druid sits between PowerDrill and Dremel on the spectrum of functionality. It can do almost anything Dremel does (except that Dremel handles

[6] Beyond Hadoop: Fast Ad-Hoc Queries on Big Data; "http://www.youtube.com/watch?v=eCbXoGSyHbg"
[7] https://github.com/metamx/druid/wiki/Design

arbitrary nested structures whereas Druid only allows for single level of array nesting) and utilizes some data layout and compression methods from PowerDrill. See below [8]:



Ultimately, Druid is *ideal* for products that require real-time ingestion of a single, large data stream – especially if you are dealing with a no-downtime operation and are building your product on top of a time-oriented summarization of the incoming data stream. If you care about query flexibility and raw data access, Druid is not right for you. Druid is fast – queries that run in single-digit seconds over a 6 TB data set, but this is limited to certain types of queries and specific types of data sets (inflexible) [9].

As for Metamarkets' use case, their online advertising customers had data volumes upwards billions of events per month. Thus, they needed two main functions: the ability to perform highly interactive queries on the latest data and the ability to arbitrarily filter across any dimension (with data sets upwards thirty dimensions). A sample query would be "find me how many advertisements were seen by female executives, aged 35 to 44, from the US, UK, and Canada reading sports blogs on the weekends." Recently, Netflix has been testing Druid for operational monitoring of real-time metrics across their streaming business. Sudhir Tonse of Netflix says, "Netflix manages billions of streaming events each day, so we need a highly scalable data store for operational reporting. We are so far impressed with the speed and

[8] Beyond Hadoop: Fast Ad-Hoc Queries on Big Data; "http://www.youtube.com/watch?v=eCbXoGSyHbg"
[9] https://github.com/metamx/druid/wiki/Design

scalability of Druid, and are continuing to evaluate it for providing critical real-time transparency into our operational metrics" [10].

When asked about the best use cases for Druid, Eric Tschetter – the lead architect of Druid – replied, "It's very suited for slice-n-dice exploration of event streams. It is also hopefully potentially useful in other areas as well, but the thing we use it for is largely human-driven data exploration." Ultimately, Druid's functionality is that of any OLAP data store [11].

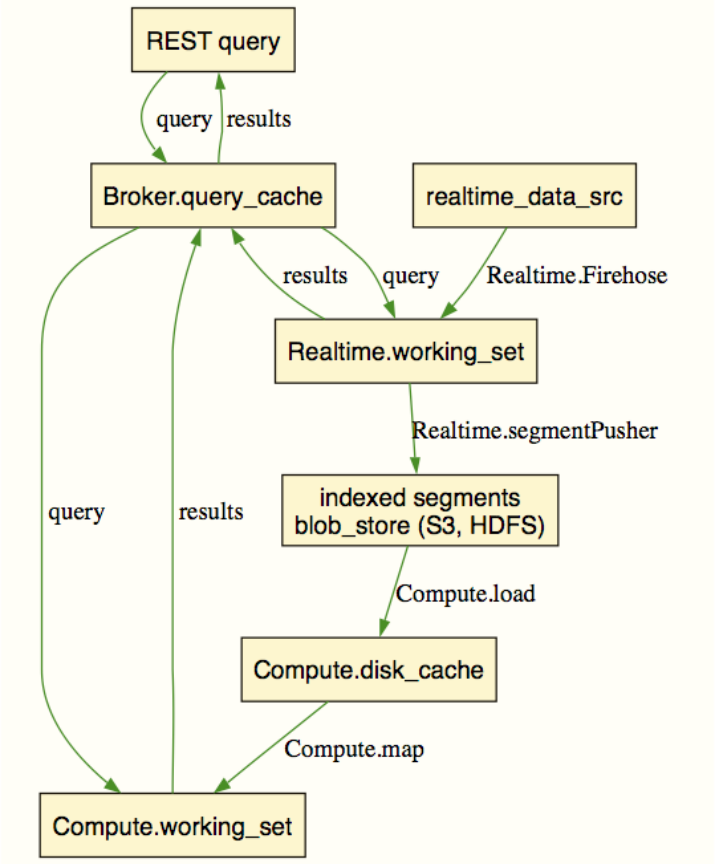
IV. Querying

A look at the main query features [12]:

Queries first enter the Broker, where the broker will match the query with the data segments that are known to exist. The Broker will then pick a set of machines serving those segments and rewrite the query for each server to specify the segments targeted. The Compute/Realtime nodes will take in the query, process them, and then return the results. The Broker takes those results and merges them together to get a final answer. This way, the Broker can prune all the data that doesn't match a query before ever looking at a row of data. For more granular filters, the indexing structures in each segment allows the compute nodes to figure out

[10] http://metamarkets.com/2012/metamarkets-open-sources-druid/
[11] https://groups.google.com/forum/#!topic/druid-development/4JmvKEEq87w
[12] Beyond Hadoop: Fast Ad-Hoc Queries on Big Data; "http://www.youtube.com/watch?v=eCbXoGSyHbg"

which rows match the filter set before examining the data (done through Boolean algebra) [13]. See the below diagram (batch indexing not shown) [14]:



Queries are made using an HTTP REST style request. The query is expressed in JSON. Here is an example query from the Druid wiki [15]:

[13] https://github.com/metamx/druid/wiki/Design
[14] https://github.com/metamx/druid/wiki
[15] https://github.com/metamx/druid/wiki/Querying

```
{
    "queryType": "groupBy",
    "dataSource": "randSeq",
    "granularity": "all",
    "dimensions": [],
    "aggregations": [
        { "type": "count", "name": "rows" },
        { "type": "doubleSum", "fieldName": "events", "name": "e" },
        { "type": "doubleSum", "fieldName": "outColumn", "name": "randomNumberSum" }
    ],
    "postAggregations": [{
       "type": "arithmetic",
       "name": "avg_random",
       "fn": "/",
       "fields": [
            { "type": "fieldAccess", "fieldName": "randomNumberSum" },
            { "type": "fieldAccess", "fieldName": "rows" }
       ]
    }],
    "intervals": ["2012-10-01T00:00/2020-01-01T00"]
}
```

1. queryType – identifies which kind of query operator to be used (ex: groupBy, search, timeBoundary)

2. dataSource – identifies where to apply the query (in this case, randSeq corresponds to examples/rand/rand_realtime.spec file schema). Here is a sample .spec file:

```
[{
    "schema": {
        "dataSource": "randseq",
        "aggregators": [
            {"type": "count", "name": "events"},
            {"type": "doubleSum", "name": "outColumn", "fieldName": "inColumn"}
        ],
        "indexGranularity": "minute",
        "shardSpec": {"type": "none"}
    },

    "config": {
        "maxRowsInMemory": 50000,
        "intermediatePersistPeriod": "PT10m"
    },

    "firehose": {
        "type": "rand",
        "sleepUsec": 100000,
        "maxGeneratedRows": 5000000,
        "seed": 0,
        "nTokens": 19,
        "nPerSleep": 3
    },

    "plumber": {
        "type": "realtime",
        "windowPeriod": "PT5m",
        "segmentGranularity": "hour",
        "basePersistDirectory": "/tmp/rand_realtime/basePersist"
    }
}]
```

3. granularity – specifies bucket size for values (ex: second, minute, all)
4. dimensions – array of zero or more fields as defined in the dataSource spec file or defined in the input records. Used to constrain the grouping. If empty, then one value per time granularity bucket is requested in the groupBy
5. aggregations – this is required by groupBy queries. Aggregations are applied to the column specified by fieldname and the output of the aggregation will be named according to the value in the name field.
6. intervals – the time range of the query. Data outside the specified intervals will not be used; this example specifies from Oct 1, 2012 till Jan 1, 2020

Here is a full table detailing query operators for various queries [16]:

query types	property	description	required?
timeseries, groupBy, search, timeBoundary	dataSource	query is applied to this data source	yes
timeseries, groupBy, search, timeBoundary	intervals	range of time series to include in query	yes
timeseries, groupBy, search, timeBoundary	context	This is a key-value map that can allow the query to alter some of the behavior of a query. It is primarily used for debugging, for example if you include `"bySegment":true` in the map, you will get results associated with the data segment they came from.	no
timeseries, groupBy, search	filter	Specifies the filter (the "WHERE" clause in SQL) for the query. See Filters	no
timeseries, groupBy, search	granularity	the timestamp granularity to bucket results into (i.e. "hour"). See Granularities for more information.	no
timeseries, groupBy	dimensions	constrains the groupings; if empty, then one value per time granularity bucket	yes
timeseries, groupBy	aggregations	aggregations that combine values in a bucket. See Aggregations.	yes
timeseries, groupBy	postAggregations	aggregations of aggregations. See Post Aggregations.	yes
search	limit	maximum number of results (default is 1000), a system-level maximum can also be set via `com.metamx.query.search.maxSearchLimit`	no

[16] https://github.com/metamx/druid/wiki/Querying

search	searchDimensions	Dimensions to apply the search query to. If not specified, it will search through all dimensions.	no
search	query	The query portion of the search query. This is essentially a predicate that specifies if something matches.	yes