Druid: Real-time Analytical Data Store

Fangjin Yang
Metamarkets, Inc.
625 2nd Street, Suite 230
San Francisco, CA, USA
fangjin@metamarkets.com

Eric Tschetter
Metamarkets, Inc.
625 2nd Street, Suite 230
San Francisco, CA, USA
eric@metamarkets.com

Gian Merlino
Metamarkets, Inc.
625 2nd Street, Suite 230
San Francisco, CA, USA
gian@metamarkets.com

Nelson Ray
Metamarkets, Inc.
625 2nd Street, Suite 230
San Francisco, CA, USA
nelson@metamarkets.com

Xavier Léauté
Metamarkets, Inc.
625 2nd Street, Suite 230
San Francisco, CA, USA
xavier@metamarkets.com

Deep Ganguli
Metamarkets, Inc.
625 2nd Street, Suite 230
San Francisco, CA, USA
deep@metamarkets.com

ABSTRACT

Druid is a scalable, real-time analytical data store that supports ad-hoc queries on large-scale data sets. The system combines a columnar data layout, a shared-nothing architecture, and an advanced indexing structure to allow arbitrary exploration of billion-row tables with sub-second latencies. Druid scales horizontally and is the core engine of the Metamarkets platform. In this paper, we detail the architecture and implementation of Druid and describe how it solves the real-time data ingestion problem.

1. INTRODUCTION

In recent years, enterprises are facing ever-growing collections of data in all forms. The scale of data has reached terabytes, even petabytes of information per day. Companies are increasingly realizing the importance of unlocking the insights contained within this data. Numerous database vendors such as IBM's Netezza [31], Vertica [8], and EMC's Greenplum [28] offer data warehousing solutions, and several research papers [7, 11, 14] directly address this problem as well. As the interactive data exploration space becomes more mature, it is apparent that real-time ingestion and exploration of data will unlock business-critical decisions for front office and back office analysts.

Metamarkets realized early on that for high data volumes, existing Relational Database Management Systems (RDBMS) and most NoSQL architectures were not sufficient to address the performance and use-case needs of the business intelligence space. Druid was built to address a gap we believe exists in the current big-data ecosystem for a real-time analytical data store. Druid is a distributed, columnar, shared-nothing data store designed to reliably scale to petabytes of data and thousands of cores. It is a highly available system designed to run on commodity hardware with no downtime in the face of failures, data imports or software updates. We have been building Druid over the course of the last two years and it is the core engine of the Metamarkets technology stack.

In many ways, Druid shares similarities with other interactive query systems [27], main-memory databases [16], and widely-known distributed data stores such as BigTable [10], Dynamo [13], and Cassandra [23]. Unlike most traditional data stores, Druid operates mainly on read-only data and has limited functionality for writes. The system is highly optimized for large-scale transactional data aggregation and arbitrarily deep data exploration. Druid is highly configurable and allows users to adjust levels of fault tolerance and performance.

Druid builds on the ideas of other distributed data stores, real-time computation engines, and search engine indexing algorithms. In this paper, we make the following contributions to academia:

- We outline Druid's real-time ingestion and query capabilities and explain how we can explore events within milliseconds of their creation
- We describe how the architecture allows for fast and flexible queries and how inverted indices can be applied to quickly filter data
- We present experiments benchmarking Druid's performance

The format of the paper is as follows: Section 2 describes the Druid data model. Section 3 presents an overview of the components of a Druid cluster. Section 4 outlines the query API. Section 5 describes data storage format in greater detail. Section 6 discusses Druid robustness and failure responsiveness. Section 7 provides our performance benchmarks. Section 8 lists the related work and Section 9 presents our conclusions.

2. DATA MODEL

The fundamental storage unit in Druid is the segment. Distribution and replication in Druid are always done at a segment level. Segments encapsulate a partition of a larger transactional data set. To better understand how a typical row/column collection of data translates into a Druid segment, consider the data set shown in Table 1.

A segment is composed of multiple binary files, each representing a column of a data set. The data set in Table 1 consists of 8 distinct columns, one of which is the timestamp column. Druid always requires a timestamp column because it (currently) only operates with event-based data. Segments always represent some time interval and each column file contains the specific values for that column over the time interval. Since segments always contain data

Table 1: Sample Druid data

Timestamp	Publisher	Advertiser	Gender	Country	Impressions	Clicks	Revenue
2011-01-01T01:00:00Z	bieberfever.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	USA	1953	17	17.31
2011-01-01T02:00:00Z	ultratrimfast.com	google.com	Male	USA	3194	170	34.01

for a time range, it is logical that Druid partitions data into smaller chunks based on the timestamp value. In other words, segments can be thought of as blocks of data that represent a certain granularity of time. For example, if we wanted to shard the data in Table 1 to an hourly granularity, the partitioning algorithm would result in two segments, one representing each hour of 2011-01-01. Similarly, if we sharded the data to a daily granularity, we would create a single segment for 2011-01-01.

Partitioning the data based on granularity buckets allows users to fine tune the degree of parallelization they want in Druid. A data set representing a year's worth of data may be bucketed by day, and a data set representing only a day's worth of day may be partitioned by hour. Sharding on a single dimension (time) may still result in segments that are too large to manage if the data volume is sufficiently high. To create more operable partition chunks, Druid may additionally shard data based on other factors such as dimension cardinality. Each shard creates a segment and hence, segments are uniquely identified by a data source id describing the data, the time interval of the data, a version indicating when the segment was created, and a shard partition number.

Data in Druid can be conceptually thought of as being either real-time or historical. Real-time data refers to recently ingested data; typically, in a production setting, this will be data for the current hour. Historical data refers to any data that is older. Segments can be similarly classified as being either real-time or historical.

Historical segments are immutable and do not support insert, delete, or update semantics. By maintaining immutable blocks of data within the system, we can maintain a consistent snapshot of historical data and provide read consistency without having to worry about concurrent updates and deletes. If updates to a historical segment are required, we build a new segment for the same data source and time interval with the updated data. This new segment will have an updated version identifier.

Multiple segments for the same data source and time range may exist in the system at any time. To provide read consistency, Druid read operations always access data in a particular time range from the segment with the latest version identifier for that time range. Historical segments may be stored on local disk or in a key-value "deep" store such as S3 [13] or HDFS [30]. All historical segments have associated metadata describing properties of the segment such as size in bytes, compression format, and location in deep storage.

Real-time segments are mutable and generally represent a much shorter duration of time than historical segments. Real-time segments contain recent data and are incrementally populated as new events are ingested. On a periodic basis, real-time segments are converted into historical segments. Additional details about this conversion process are given in Section 3.3.

3. CLUSTER

A Druid cluster consists of different types of nodes, each performing a specific function. The composition of a Druid cluster is shown in Figure 1.

Recall that data in Druid is classified as either real-time or historical. The Druid cluster is architected to reflect this conceptual separation of data. Real-time nodes are responsible for ingesting, storing, and responding to queries for the most recent events. Similarly, historical compute nodes are responsible for loading and responding to queries for historical events.

Data in Druid is stored on storage nodes. Storage nodes can be either compute or real-time nodes. Queries to access this data will typically first hit a layer of broker nodes. Broker nodes are responsible for finding and routing queries down to the storage nodes that host the pertinent data. The storage nodes compute their portion of the query response in parallel and return their results to the brokers. Broker nodes, compute nodes, and realtime nodes can all be classified as queryable nodes.

Druid also has a set of coordination nodes to manage load assignment, distribution, and replication. Coordination nodes are not queryable and instead focus on maintaining cluster stability. Coordination nodes have an external dependency on a MySQL database.

Druid relies on an Apache Zookeeper [20] cluster for coordination. This dependency is required because there is no direct coordination-related communication between Druid nodes. The following sections will discuss each Druid component in greater detail.

3.1 Apache Zookeeper

Zookeeper is a service for coordinating processes of distributed applications. Zookeeper provides connecting applications an abstraction of a hierarchy of data nodes known as znodes. Each znode is part of a hierarchical namespace, similar to file systems. Zookeeper has the concept of ephemeral and permanent znodes. Permanent nodes must be created and destroyed explicitly by a connecting application. Ephemeral znodes can be created by connecting applications and deleted either explicitly or if the session that created the znode is terminated (such as in the event of service failure).

3.2 Historical Compute Nodes

Historical compute nodes are the main workers of a Druid cluster and are self-contained and self-sufficient. Compute nodes load historical segments from permanent/deep storage and expose them for querying. There is no single point of contention between the nodes and nodes have no knowledge of one another. Compute nodes are operationally simple; they only know how to perform the tasks they are assigned. To help other services discover compute nodes and the data they hold, every compute node maintains a constant Zookeeper connection. Compute nodes announce their online state and the segments they serve by creating ephemeral nodes under specifically configured Zookeeper paths. Instructions for a given compute node to load new segments or drop existing segments are sent by creating ephemeral znodes under a special "load queue" path associated with the compute node.

To expose a segment for querying, a compute node must first possess a local copy of the segment. Before a compute node downloads a segment from deep storage, it first checks a local disk directory (cache) to see if the segment already exists in local storage. If no cache information about the segment is present, the compute node will download metadata about the segment from Zookeeper. This metadata includes information about where the segment is located

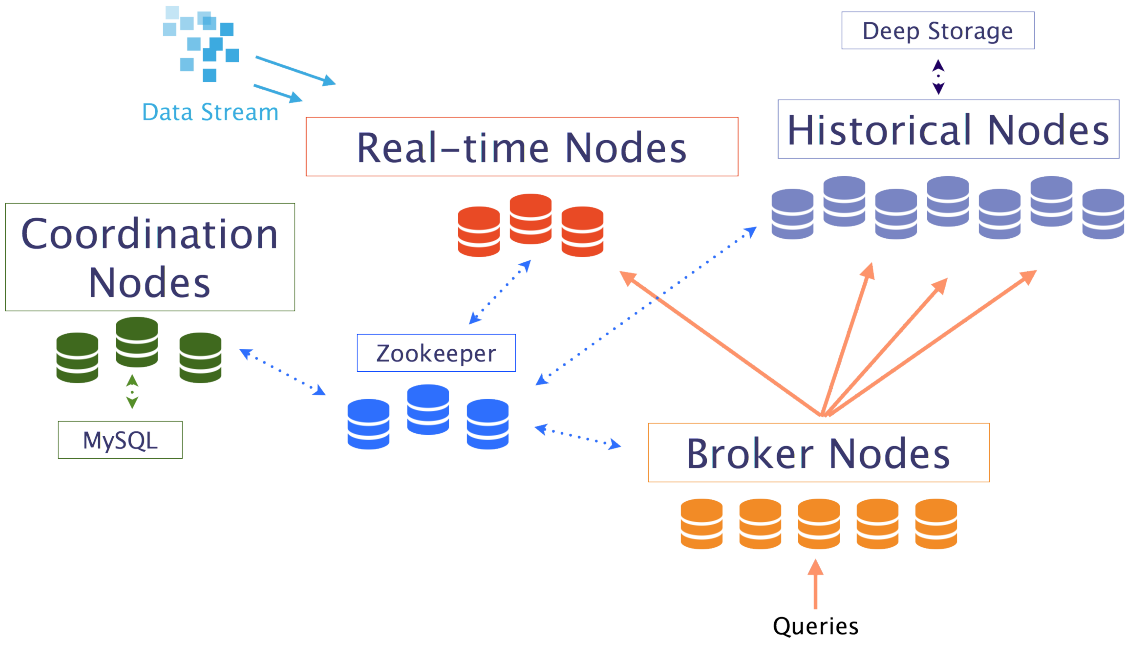

Figure 1: An overview of a Druid cluster.

in deep storage and about how to decompress and process the segment. Once a compute node completes processing a segment, the node announces (in Zookeeper) that it is serving the segment. At this point, the segment is queryable.

3.2.1 Tiers

Compute nodes can be grouped in different tiers, where all nodes in a given tier are identically configured. Different performance and fault-tolerance parameters can be set for each tier. The purpose of tiered nodes is to enable higher or lower priority segments to be distributed according to their importance. For example, it is possible to spin up a "hot" tier of compute nodes that have a high number of cores and a large RAM capacity. The "hot" cluster can be configured to download more frequently accessed segments. A parallel "cold" cluster can also be created with much less powerful backing hardware. The "cold" cluster would only contain less frequently accessed segments.

3.3 Real-time Nodes

Real-time nodes encapsulate the functionality to ingest and query real-time data streams. Data indexed via these nodes is immediately available for querying. Real-time nodes are a consumer of data and require a corresponding producer to provide the data stream. Typically, for data durability purposes, a message bus such as Kafka [22] sits between the producer and the real-time node as shown in Figure 2.

The purpose of the message bus in Figure 2 is to act as a buffer for incoming events. The message bus can maintain offsets indicating the position in an event stream that a real-time node has read up to and real-time nodes can update these offsets periodically. The message bus also acts as a backup storage for recent events. Real-time nodes ingest data by reading events from the message bus. The time from event creation to message bus storage to event consumption is on the order of hundreds of milliseconds.

Real-time nodes maintain an in-memory index for all incoming events. These indexes are incrementally populated as new events

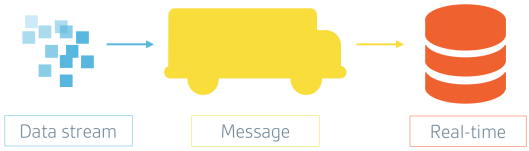

Figure 2: Real-time data ingestion.

appear on the message bus. The indexes are also directly queryable. Real-time nodes persist their indexes to disk either periodically or after some maximum row limit is reached. After each persist, a real-time node updates the message bus with the offset of the last event of the most recently persisted index. Each persisted index is immutable. If a real-time node fails and recovers, it can simply reload any indexes that were persisted to disk and continue reading the message bus from the point the last offset was committed. Periodically committing offsets reduces the number of messages a real-time node has to rescan after a failure scenario.

Real-time nodes maintain a consolidated view of the currently updating index and of all indexes persisted to disk. This unified view allows all indexes on a node to be queried. On a periodic basis, the nodes will schedule a background task that searches for all persisted indexes of a data source. The task merges these indexes together and builds a historical segment. The nodes will upload the segment to deep storage and provide a signal for the historical compute nodes to begin serving the segment. The ingest, persist, merge, and handoff steps are fluid; there is no data loss as a real-time node converts a real-time segment to a historical one. Figure 3 illustrates the process.

Similar to compute nodes, real-time nodes announce segments in Zookeeper. Unlike historical segments, real-time segments may represent a period of time that extends into the future. For example, a real-time node may announce it is serving a segment that contains data for the current hour. Before the end of the hour, the real-time

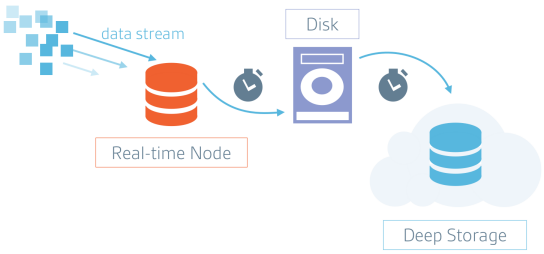

Figure 3: Real-time data durability

node continues to collect data for the hour. Every 10 minutes (the persist period is configurable), the node will flush and persist its in-memory index to disk. At the end of the current hour, the real-time node prepares to serve data for the next hour by creating a new index and announcing a new segment for the next hour. The node does not immediately merge and build a historical segment for the previous hour until after some window period has passed. Having a window period allows for straggling data points to come in and minimizes the risk of data loss. At the end of the window period, the real-time node will merge all persisted indexes, build a historical segment for the previous hour, and hand the segment off to historical nodes to serve. Once the segment is queryable on the historical nodes, the real-time node flushes all information about the segment and unannounces it is serving the segment.

Real-time nodes are highly scalable. If the data volume and ingestion rates for a given data source exceed the maximum capabilities of a single node, additional nodes can be added. Multiple nodes can consume events from the same stream, and every individual node only holds a portion of the total number of events. This creates natural partitions across nodes. Each node announces the real-time segment it is serving and each real-time segment has a partition number. Data from individual nodes will be merged at the Broker level. To our knowledge, the largest production level real-time Druid cluster is consuming approximately 2 TB of raw data per hour.

3.4 Broker Nodes

Broker nodes act as query routers to other queryable nodes such as compute and real-time nodes. Broker nodes understand the metadata published in Zookeeper about what segments exist and on what nodes the segments are stored. Broker nodes route incoming queries such that the queries hit the right storage nodes. Broker nodes also merge partial results from storage nodes before returning a final consolidated result to the caller. Additionally, brokers provide an extra level of data durability as they maintain a cache of recent results. In the event that multiple storage nodes fail and all copies of a segment are somehow lost, it is still possible that segment results can still be returned if that information exists in the cache.

3.4.1 Timeline

To determine the correct nodes to forward queries to, Broker nodes first build a view of the world from information in Zookeeper. Recall that Druid uses Zookeeper to maintain information about all compute and real-time nodes in a cluster and the segments those nodes are serving. For every data source in Zookeeper, the Broker node builds a timeline of segments for the data source and the nodes that serve them. A timeline consists of segments and represents which segments contain data for what ranges of time. Druid may have multiple segments where the data source and interval are the same but versions differ. The timeline view will always surface segments with the most recent version identifier for a time range. If two segments intervals overlap, the segment with the more recent version always has precedence. When queries are received for a specific data source and interval, the Broker node performs a lookup on the timeline associated with the query data source for the query interval and retrieves the segments that contain data for the query. The broker node maps these segments to the storage nodes that serve them and forwards the query down to the respective nodes.

3.4.2 Caching

Broker nodes employ a distributed cache with a LRU [29, 21] cache invalidation strategy. The broker cache stores per segment results. The cache can be local to each broker node or shared across multiple nodes using an external distributed cache such as memcached [18]. Recall that each time a broker node receives a query, it first maps the query to a set of segments. A subset of these segment results may already exist in the cache and the results can be directly pulled from the cache. For any segment results that do not exist in the cache, the broker node will forward the query to the compute nodes. Once the compute nodes return their results, the broker will store those results in the cache. Real-time segments are never cached and hence requests for real-time data will always be forwarded to real-time nodes. Real-time data is perpetually changing and caching the results would be unreliable.

3.5 Coordination (Master) Nodes

The Druid coordination or master nodes are primarily in charge of segment management and distribution. The Druid master is responsible for loading new segments, dropping outdated segments, managing segment replication, and balancing segment load. Druid uses a multi-version concurrency control swapping protocol for managing segments in order to maintain stable views.

The Druid master runs periodically to determine the current state of the cluster. It makes decisions by comparing the expected state of the cluster with the actual state of the cluster at the time of the run. As with all Druid nodes, the Druid master maintains a connection to Zookeeper for current cluster information. The master also maintains a connection to a MySQL database that contains additional operational parameters and configurations. One of the key pieces of information located in the MySQL database is a segment table that contains a list of historical segments that should be served. This table can be updated by any service that creates historical segments. The MySQL database also contains a rule table that governs how segments are created, destroyed, and replicated in the cluster.

The master does not directly communicate with a compute node when assigning it work; instead the master creates an ephemeral znode in Zookeeper containing information about what the compute node should do. The compute node maintains a similar connection to Zookeeper to monitor for new work.

3.5.1 Rules

Historical segments are loaded and dropped from the cluster based on a set of rules. Rules indicate how segments should be assigned to different compute node tiers and how many replicates of a segment should exist in each tier. Rules may also indicate when segments should be dropped entirely from the cluster. The master loads a set of rules from a rule table in the MySQL database. Rules may be specific to a certain data source and/or a default set of rules can be configured. The master will cycle through all available segments and match each segment with the first rule that applies to it.

3.5.2 Load Balancing

In a typical production environment, queries often hit dozens or even hundreds of data segments. Since each compute node has limited resources, historical segments must be distributed among the cluster to ensure that the cluster load is not too imbalanced. Determining optimal load distribution requires some knowledge about query patterns and speeds. Typically, queries cover recent data spanning contiguous time intervals for a single data source. On average, queries that access smaller segments are faster.

These query patterns suggest replicating recent historical segments at a higher rate, spreading out large segments that are close in time to different compute nodes, and co-locating segments from different data sources. To optimally distribute and balance segments among the cluster, we developed a cost-based optimization procedure that takes into account the segment data source, recency, and size. The exact details of the algorithm are beyond the scope of this paper.

4. QUERY API

Queries to Druid are made in the form of POST requests. All queryable Druid nodes share the same query API. The body of the POST request is a JSON object containing key-value pairs specifying various query parameters. A typical query will contain the data source name, the granularity of the result data, the time range of interest, and the type of request. A sample time series query is shown below:

```
{
  "queryType"     : "timeseries",
  "dataSource"    : "sample_data",
  "intervals"     : "2013-01-01/2013-01-02",
  "filter"        : {
    "type"        : "selector",
    "dimension"   : "poets",
    "value"       : "Ke$ha"
  },
  "granularity"   : "day",
  "aggregations"  : [
    {
      "type"      : "count",
      "fieldName" : "row",
      "name"      : "row"
    }
  ]
}
```

Certain query types will also support a filter set. A filter set is an arbitrary Boolean expression of dimension name and value pairs. Support for complex nested filter sets enables flexibility and provides the ability to deeply explore data.

The exact query syntax depends on the query type and the information requested. It is beyond the scope of this paper to describe the Query API in full detail. We are also in the process of building out SQL support for Druid.

5. STORAGE

Druid is a column-oriented data store. When considering aggregates over a large number of events, the advantages storing data as columns rather than rows are well documented [9]. Column storage allows for more efficient CPU usage as only what is needed is actually loaded and scanned. In a row oriented data store, all columns associated with a row must be scanned as part of an aggregation. The additional scan time can introduce performance degradations as high as 250% [8].

5.1 Column Types

Druid has multiple column types to represent the various column value formats. Depending on the column type, different compression methods are used to reduce the cost of storing a column in memory and on disk. In the example given in Table 1, the publisher, advertiser, gender, and country columns only contain strings. String columns can be dictionary encoded. Dictionary encoding is a common method to compress data and has been used in other data stores such as Powerdrill [19]. In the example in Table 1, we can map each publisher to an unique integer identifier.

```
bieberfever.com   -> 0
ultratrimfast.com -> 1
```

This mapping allows us to represent the publisher column as an integer array where the array indices correspond to the rows of the original data set. For the publisher column, we can represent the unique publishers as follows:

```
[0, 0, 1, 1]
```

The resulting integer array lends itself very well to compression. Generic compression algorithms on top of encodings are very common in column-stores. We opted to use the LZF [2] compression algorithm.

We can leverage similar compression algorithms for numeric columns. For example, the clicks and revenue columns in Table 1 can also be expressed as individual arrays.

```
Clicks  -> [25, 42, 17, 170]
Revenue -> [15.70, 29.18, 17.31, 34.01]
```

In this case we compress the raw values as opposed to their dictionary representations.

5.2 Filters

To support arbitrary filter sets, Druid creates additional lookup indices for string columns. These lookup indices are compressed and Druid operates over the indices in their compressed form. Filters can be expressed as Boolean equations of multiple lookup indices. Boolean operations of indices in their compressed form is both performance and space efficient.

Let us consider the publisher column in Table 1. For each unique publisher in Table 1, we can form some representation indicating which table rows a particular publisher is seen. We can store this information in a binary array where the array indices represent our rows. If a particular publisher is seen in a certain row, that array index is marked as `1`. For example:

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

`bieberfever.com` is seen in rows `0` and `1`. This mapping of column values to row indices forms an inverted index [33]. If we want to know which rows contain `bieberfever.com` or `ultratrimfast.com`, we can `OR` together the `bieberfever.com` and `ultratrimfast.com` arrays.

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

This approach of performing Boolean operations on large bitmap sets is commonly used in search engines. Bitmap compression algorithms are a well-defined area of research and often utilize run-length encoding. Well known algorithms include Byte-aligned Bitmap Code [6], Word-Aligned Hybrid (WAH) code [36], and Partitioned Word-Aligned Hybrid (PWAH) compression [34]. Druid opted to

Figure 4: Integer array size versus concise set size.

use the CONCISE algorithm [12] as it can outperform WAH by reducing the size of the compressed bitmaps by up to 50%. Figure 4 illustrates the number of bytes using CONCISE compression versus using an integer array. The results were generated on a cc2.8xlarge system with a single thread, 2G heap, 512m young gen, and a forced GC between each run. The data set is a single day's worth of data collected from the Twitter garden hose [4] data stream. The data set contains 2,272,295 rows and 12 dimensions of varying cardinality. As an additional comparison, we also resorted the data set rows to maximize compression.

In the unsorted case, the total CONCISE compressed size was 53,451,144 bytes and the total integer array size was 127,248,520 bytes. Overall, CONCISE compressed sets are about 42% smaller than integer arrays. In the sorted case, the total CONCISE compressed size was 43,832,884 bytes and the total integer array size was 127,248,520 bytes. What is interesting to note is that after sorting, global compression only increased minimally. The total CONCISE set size to total integer array size is 34%. It is also interesting to note that as the cardinality of a dimension approaches the total number of rows in a data set, integer arrays actually require less space than CONCISE sets.

5.3 Storage Engine

Druid's persistence components allows for different storage engines to be plugged in, similar to Dynamo [13]. These storage engines may store data in in-memory structures such as the JVM heap or in memory mapped structures. The ability to swap storage engines allows for Druid to be configured depending on a particular application's specifications. An in-memory storage engine may be operationally more expensive than a memory-mapped storage engine but could be a better alternative if performance is critical. At Metamarkets, we commonly use a memory-mapped storage engine.

6. ROBUSTNESS AND FAULT-TOLERANCE

To achieve high system availability and data durability, Druid employs several fault recovery techniques. Druid has no single point of failure.

6.1 Replication

Druid replicates historical segments on multiple hosts. The number of replicates in each tier of the historical compute cluster is fully configurable. Setups that require high levels of fault tolerance can be configured to have a high number of replicates. Replicates are assigned to compute nodes by coordination nodes using the same load distribution algorithm discussed in Section 3.4.2. Conceptually, broker nodes do not distinguish historical segments from their replicates. Broker nodes forward queries to the first node it finds that contains data for the query.

Real-time segments follow a different replication model as real-time segments are mutable. Recall that real-time nodes act as consumers of a data stream. Multiple real-time nodes can read from the same message bus if each maintains a unique offset, hence creating multiple copies of a real-time segment. If a real-time node fails and recovers, it can reload any indexes that were persisted to disk and read from the message bus from the point it last committed an offset.

6.2 Local Segment Cache

Recall that each Druid compute node maintains a local cache of historical segments it recently served. A compute node also has a lookup table for segments it has in its cache and stores this lookup table on disk. When a compute node is assigned a new segment to load, the compute node will first check its local segment cache directory to see if the segment had been previously downloaded. If a cache entry exists, the compute node will directly read the segment binary files and load the segment.

The segment cache is also leveraged when a compute node is initially started. During startup, a compute node will first read its lookup table to determine what segments it has cached. All cached segments are immediately loaded and served. This feature introduces minimal overhead and allows a compute node to readily serve data as soon as it comes online. By making data quickly available on startup and minimizing initial startup time, compute nodes that become inexplicably disconnected from the cluster can reconnect themselves seamlessly. This also means that software updates can be pushed to compute nodes without disruption to cluster operations. In practice, a software update to a compute node can be completed before coordination nodes even notice that the compute node has disconnected. At Metamarkets, we update Druid through rolling restarts. Compute nodes are updated one at a time and we experience no downtime or data loss through the update process.

6.3 Failure Detection

If a compute node completely fails and becomes unavailable, the ephemeral Zookeeper znodes it created are deleted. The master node will notice that certain segments are insufficiently replicated or missing altogether. Additional replicates will be created and redistributed throughout the cluster.

We are moving towards building out infrastructure to support programmatic creation of real-time nodes. In the near future, the master node will also notice if real-time segments are insufficiently replicated and automatically create additional real-time nodes as redundant backups.

6.4 Adding and Removing Nodes

Starting and removing Druid nodes is a relatively simple process; all that is required is to start and stop Java processes. There is minimal operational overhead with adding nodes in batches. Scaling down the cluster is usually done one node at a time with some time lapse between shutdowns. This allows the master to have ample time to redistribute load and create additional replicates. Shutting down nodes in batches is generally not supported as it may destroy all copies of a segment, which would lead to data loss.

7. PERFORMANCE BENCHMARKS

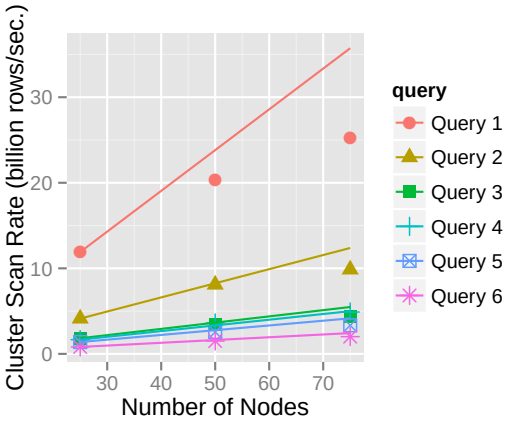

Figure 5: Druid cluster scan rate with lines indicating linear scaling from 25 nodes.

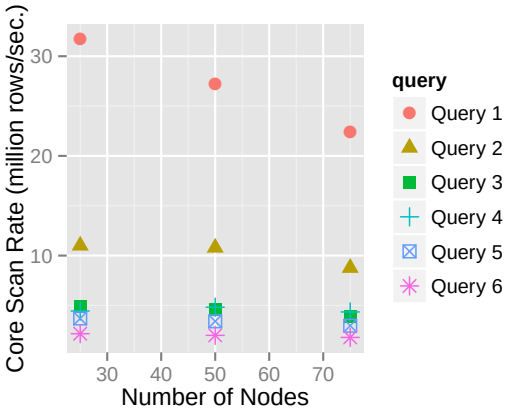

Figure 6: Druid core scan rate.

To benchmark Druid performance, we created a large test cluster with 6TB of uncompressed data, representing tens of billions of fact rows. The data set contained more than a dozen dimensions, with cardinalities ranging from the double digits to tens of millions. We computed four metrics for each row (counts, sums, and averages). The data was sharded first on timestamp then on dimension values, creating thousands of shards roughly 8 million fact rows apiece.

The cluster used in the benchmark consisted of 100 historical compute nodes, each with 16 cores, 60GB of RAM, 10 GigE Ethernet, and 1TB of disk space. Collectively, the cluster comprised of 1600 cores, 6TB or RAM, fast Ethernet and more than enough disk space.

SQL statements are included in Table 2 to describe the purpose of each of the queries. Please note:

- The timestamp range of the queries encompassed all data.
- Each machine was a 16-core machine with 60GB RAM and 1TB of local disk. The machine was configured to only use 15 threads for processing queries.
- A memory-mapped storage engine was used (the machine was configured to memory map the data instead of loading it into the Java heap.)

Figure 5 shows the cluster scan rate and Figure 6 shows the core scan rate. In Figure 5 we also include projected linear scaling based on the results of the 25 core cluster. In particular, we observe diminishing marginal returns to performance in the size of the cluster. Under linear scaling, SQL query 1 would have achieved a speed of 37 billion rows per second on our 75 node cluster. In fact, the speed was 26 billion rows per second. The speed up of a parallel computing system is often limited by the time needed for the sequential operations of the system, in accordance with Amdahl's law [5].

The first query listed in Table 2 is a simple count, achieving scan rates of 33M rows/second/core. We believe the 75 node cluster was actually overprovisioned for the test dataset, explaining the modest improvement over the 50 node cluster. Druid's concurrency model is based on shards: one thread will scan one shard. If the number of segments on a compute node modulo the number of cores is small (e.g. 17 segments and 15 cores), then many of the cores will be idle during the last round of the computation.

When we include more aggregations we see performance degrade. This is because of the column-oriented storage format Druid employs. For the `count(*)` queries, Druid only has to check the timestamp column to satisfy the "where" clause. As we add metrics, it has to also load those metric values and scan over them, increasing the amount of memory scanned.

8. RELATED WORK

Cattell [9] maintains a great summary about existing Scalable SQL and NoSQL data stores. In the landscape of distributed data stores, Druid feature-wise sits somewhere between Google's Dremel [27] and Powerdrill [19]. Druid has most of the features implemented in Dremel (Dremel handles arbitrary nested data structures while Druid only allows for a single level of array-based nesting) and many of the interesting compression algorithms mentioned in Powerdrill.

Although Druid builds on many of the same principles as other distributed column-oriented data stores [17], most existing data stores are designed to be key-value stores [24], or document/extensible record stores [32]. Such data stores are great solutions for traditional data warehouse needs and general back-office/reporting usage. Typically, analysts will query these data stores and build reports from the results. In-memory databases such as SAP's HANA [16] and VoltDB [35] are examples of other data stores that are highly suited for traditional data warehousing needs. Druid is a front-office system designed such that user-facing dashboards can be built on top of it. Similar to [3], Druid has analytical features built in. The main features Druid offers over traditional data warehousing solutions are real-time data ingestion, interactive queries and interactive query latencies. In terms of real-time ingestion and processing of data, Trident/Storm [26] and Streaming Spark [37] are other popular real-time computation systems, although they lack the data storage capabilities of Druid. Spark/Shark [15] are also doing similar work in the area of fast data analysis on large scale data sets. Cloudera Impala [1] is another system focused on optimizing querying performance, but more so in Hadoop environments.

Druid leverages a unique combination of algorithms in its architecture. Although we believe no other data store has the same set of functionality as Druid, some of Druid's features such as using inverted indices to perform faster filters also exist in other data stores [25].

9. CONCLUSIONS

Table 2: Druid Queries

```
1 SELECT count(*) FROM _table_ WHERE timestamp >= ? AND timestamp < ?

2 SELECT count(*), sum(metric1) FROM _table_ WHERE timestamp >= ? AND timestamp < ?

3 SELECT count(*), sum(metric1), sum(metric2), sum(metric3), sum(metric4) FROM _table_
  WHERE timestamp >= ? AND timestamp < ?

4 SELECT high_card_dimension, count(*) AS cnt FROM _table_
  WHERE timestamp >= ? AND timestamp < ?
  GROUP BY high_card_dimension ORDER BY cnt limit 100

5 SELECT high_card_dimension, count(*) AS cnt, sum(metric1) FROM _table_
  WHERE timestamp >= ? AND timestamp < ?
  GROUP BY high_card_dimension ORDER BY cnt limit 100

6 SELECT high_card_dimension, count(*) AS cnt, sum(metric1), sum(metric2), sum(metric3), sum(metric4)
  FROM _table_ WHERE timestamp >= ? AND timestamp < ?
  GROUP BY high_card_dimension ORDER BY cnt limit 100
```

In this paper, we presented Druid, a distributed, column-oriented, real-time analytical data store. Druid is a highly customizable solution that is optimized for fast query latencies. Druid ingests data in real-time and is fault-tolerant. We discussed the performance of Druid on billion row data sets. We summarized key Druid architecture aspects such as the storage format, query language and general execution. In the future, we plan to cover more in depth the different algorithms we've developed for Druid and how other systems may plug into Druid to achieve powerful use cases.

10. ACKNOWLEDGEMENTS

Druid could not have been built without the help of many great engineers at Metamarkets and in the community. We want to thank Steve Harris, Jaypal Sethi, Danny Yuan, Jae Hyeon Bae, Paul Baclace, Dave Nielsen, Katherine Chu, and Dhruv Parthasarathy for their contributions to Druid.

11. ADDITIONAL AUTHORS

Michael Driscoll (Metamarkets, mike@metamarkets.com)

12. REFERENCES

[1] Cloudera impala: Real-time queries in apache hadoop, for real, March 2013.

[2] Liblzf, March 2013.

[3] Paraccel analytic database, March 2013.

[4] Public streams, March 2013.

[5] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[6] G. Antoshenkov. Byte-aligned bitmap compression. In *Data Compression Conference, 1995. DCC'95. Proceedings*, page 476. IEEE, 1995.

[7] L. A. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 4(1):1–108, 2009.

[8] C. Bear, A. Lamb, and N. Tran. The vertica database: Sql rdbms for managing big data. In *Proceedings of the 2012 workshop on Management of big data systems*, pages 37–38. ACM, 2012.

[9] R. Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[11] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[12] A. Colantonio and R. Di Pietro. Concise: Compressed 'n'composable integer set. *Information Processing Letters*, 110(16):644–650, 2010.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[14] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[15] C. Engle, A. Lupher, R. Xin, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 international conference on Management of Data*, pages 689–692. ACM, 2012.

[16] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[17] B. Fink. Distributed computation on dynamo-style distributed storage: riak pipe. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*, pages 43–50. ACM, 2012.

[18] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, (124):72–74, 2004.

[19] A. Hall, O. Bachmann, R. Büssow, S. Gănceanu, and M. Nunkesser. Processing a trillion cells per mouse click.

Proceedings of the VLDB Endowment, 5(11):1436–1446, 2012.

[20] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, volume 10, 2010.

[21] C. S. Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12), 2001.

[22] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011.

[23] A. Lakshman and P. Malik. Cassandra—a decentralized structured storage system. *Operating systems review*, 44(2):35, 2010.

[24] R. Lerner. At the forge: Redis. *Linux Journal*, 2010(197):3, 2010.

[25] R. MacNicol and B. French. Sybase iq multiplex-designed for analytics. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 1227–1230. VLDB Endowment, 2004.

[26] N. Marz. Storm: Distributed and fault-tolerant realtime computation, February 2013.

[27] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[28] D. Miner. Unified analytics platform for big data. In *Proceedings of the WICSA/ECSA 2012 Companion Volume*, pages 176–176. ACM, 2012.

[29] E. J. O'neil, P. E. O'neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. In *ACM SIGMOD Record*, volume 22, pages 297–306. ACM, 1993.

[30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[31] M. Singh and B. Leonhardi. Introduction to the ibm netezza warehouse appliance. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pages 385–386. IBM Corp., 2011.

[32] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[33] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Parallel and Distributed Information Systems, 1993., Proceedings of the Second International Conference on*, pages 8–17. IEEE, 1993.

[34] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *Proceedings of the 2011 international conference on Management of data*, pages 913–924. ACM, 2011.

[35] L. VoltDB. Voltdb technical overview, 2010.

[36] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems (TODS)*, 31(1):1–38, 2006.

[37] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10. USENIX Association, 2012.