Druid: A Real-time Analytical Data Store

Fangjin Yang, Eric Tschetter, Gian Merlino, Nelson Ray, Xavier Léauté, Deep Ganguli, Himadri Singh
{fangjin, cheddar, gian, nelson, xavier, deep, himadri}@metamarkets.com

ABSTRACT

Druid is an open source[1] data store designed for real-time exploratory analytics on large data sets. The system combines a column-oriented storage layout, a distributed, shared-nothing architecture, and an advanced indexing structure to allow for the arbitrary exploration of billion-row tables with sub-second latencies. In this paper, we describe Druid's architecture, and detail how it supports fast aggregations, flexible filters, and low latency data ingestion.

1. INTRODUCTION

In recent years, the proliferation of internet technology has created a surge in machine-generated events. Individually, these events contain minimal useful information and are of low value. Given the time and resources required to extract meaning from large collections of events, many companies were willing to discard this data instead. Although infrastructure has been built handle event based data (e.g. IBM's Netezza[32], HP's Vertica[6], and EMC's Greenplum[28]), they are largely sold at high price points and are only targeted towards those companies who can afford the offerings.

A few years ago, Google introduced MapReduce [11] as their mechanism of leveraging commodity hardware to index the internet and analyze logs. The Hadoop [31] project soon followed and was largely patterned after the insights that came out of the original MapReduce paper. Hadoop is currently deployed in many organizations to store and analyze large amounts of log data. Hadoop has contributed much to helping companies convert their low-value event streams into high-value aggregates for a variety of applications such as business intelligence and A-B testing.

As with a lot of great systems, Hadoop has opened our eyes to a new space of problems. Specifically, Hadoop excels at storing and providing access to large amounts of data, however, it does not make any performance guarantees around how quickly that data can be accessed. Furthermore, although Hadoop is a highly available system, performance degrades under heavy concurrent load. Lastly, while Hadoop works well for storing data, it is not optimized for ingesting data and making that data immediately readable.

Early on in the development of the Metamarkets product, we ran into each of these issues and came to the realization that Hadoop is a great back-office, batch processing, and data warehousing system. However, as a company that has product-level guarantees around query performance and data availability in a highly concurrent environment (1000+ users), Hadoop wasn't going to meet our needs. We explored different solutions in the space, and after trying both Relational Database Management Systems and NoSQL architectures, we came to the conclusion that there was nothing in the open source world that could be fully leveraged for our requirements.

We ended up creating Druid, an open-source, distributed, column-oriented, realtime analytical data store. In many ways, Druid shares similarities with other interactive query systems [27], main-memory databases [14], and widely-known distributed data stores such as BigTable [8], Dynamo [12], and Cassandra [22]. The distribution and query model also borrow ideas from current generation search infrastructure [24, 4, 5].

This paper describes the architecture of Druid, explores the various design decisions made in creating an always-on production system that powers a hosted service, and attempts to help inform anyone who faces a similar problem about a potential method of solving it. Druid is deployed in production at several technology companies[2].

The structure of the paper is as follows: we first describe the problem in Section 2. Next, we detail system architecture from the point of view of how data flows through the system in Section 3. We then discuss how and why data gets converted into a binary format in Section 4. We briefly describe the query API in Section 5. Lastly, we leave off with some benchmarks in Section 6, related work in Section 7 and conclusions are Section 8.

2. PROBLEM DEFINITION

Druid was originally designed to solve problems around ingesting and exploring large quantities of transactional events (log data). This form of timeseries data is commonly found in OLAP workflows and the nature of the data tends to be very append heavy. For example, consider the data shown in Table 1. Table 1 contains data for edits that have occurred on Wikipedia. Each time a user edits a page in Wikipedia, an event is generated that contains metadata about the edit. This metadata is comprised of 3 distinct components. First, there is a timestamp column indicating when the edit was made. Next, there are a set dimension columns indicating various attributes about the edit such as the page that was edited, the user who made the edit, and the location of the user. Finally, there are a set of metric columns that contain values (usually numeric) to aggregate over, such as the number of characters added or removed in an edit.

Our goal is to rapidly compute drill-downs and aggregates over this data. We want to answer questions like "How many edits were made on the page Justin Bieber from males in San Francisco?" and "What is the average number of characters that were added by people from

[1] https://github.com/metamx/druid

[2] http://druid.io/druid.html

Table 1: Sample Druid data for edits that have occurred on Wikipedia.

Timestamp	Page	Username	Gender	City	Characters Added	Characters Removed
2011-01-01T01:00:00Z	Justin Bieber	Boxer	Male	San Francisco	1800	25
2011-01-01T01:00:00Z	Justin Bieber	Reach	Male	Waterloo	2912	42
2011-01-01T02:00:00Z	Ke$ha	Helz	Male	Calgary	1953	17
2011-01-01T02:00:00Z	Ke$ha	Xeno	Male	Taiyuan	3194	170

Calgary over the span of a month?". We also want queries over any arbitrary combination of dimensions to return with sub-second latencies.

The need for Druid was facilitated by the fact that existing open source Relational Database Management Systems and NoSQL key/value stores were unable to provide a low latency data ingestion and query platform for interactive applications [35]. In the early days of Metamarkets, we were focused on building a hosted dashboard that would allow users to arbitrary explore and visualize event streams. The data store powering the dashboard needed to return queries fast enough that the data visualizations built on top of it could update provide users with an interactive experience.

In addition to the query latency needs, the system had to be multi-tenant and highly available. The Metamarkets product is used in a highly concurrent environment. Downtime is costly and many businesses cannot afford to wait if a system is unavailable in the face of software upgrades or network failure. Downtime for startups, who often do not have internal operations teams, can determine whether a business succeeds or fails.

Finally, another key problem that Metamarkets faced in its early days was to allow users and alerting systems to be able to make business decisions in "real-time". The time from when an event is created to when that event is queryable determines how fast users and systems are able to react to potentially catastrophic occurrences in their systems. Popular open source data warehousing systems such as Hadoop were unable to provide the sub-second data ingestion latencies we required.

The problems of data exploration, ingestion, and availability span multiple industries. Since Druid was open sourced in October 2012, it been deployed as a video, network monitoring, operations monitoring, and online advertising analytics platform in multiple companies.

3. ARCHITECTURE

A Druid cluster consists of different types of nodes and each node type is designed to perform a specific set of things. We believe this design separates concerns and simplifies the complexity of the system. There is minimal interaction between the different node types and hence, intra-cluster communication failures have minimal impact on data availability. The different node types operate fairly independent of each other and to solve complex data analysis problems, they come together to form a fully working system. The name Druid comes from the Druid class in many role-playing games: it is a shape-shifter, capable of taking on many different forms to fulfill various different roles in a group. The composition of and flow of data in a Druid cluster are shown in Figure 1.

3.1 Real-time Nodes

Real-time nodes encapsulate the functionality to ingest and query event streams. Events indexed via these nodes are immediately available for querying. The nodes are only concerned with events for some small time range and periodically hand off immutable batches of events they've collected over this small time range to other nodes in the Druid cluster that are specialized in dealing with batches of immutable events. Real-time nodes leverage Zookeeper [19] for coordination with the rest of the Druid cluster. The nodes announce their online state and the data they are serving in Zookeeper.

Real-time nodes maintain an in-memory index buffer for all incoming events. These indexes are incrementally populated as new events are ingested and the indexes are also directly queryable. Druid virtually behaves as a row store for queries on events that exist in this JVM heap-based buffer. To avoid heap overflow problems, real-time nodes persist their in-memory indexes to disk either periodically or after some maximum row limit is reached. This persist process converts data stored in the in-memory buffer to a column oriented storage format described in 4. Each persisted index is immutable and real-time nodes load persisted indexes into off-heap memory such that they can still be queried. Figure 2 illustrates the process.

On a periodic basis, each real-time node will schedule a background task that searches for all locally persisted indexes. The task merges these indexes together and builds an immutable block of data that contains all the events that have ingested by a real-time node for some span of time. We refer to this block of data as a "segment". During the handoff stage, a real-time node uploads this segment to a permanent backup storage, typically a distributed file system such as S3 [12] or HDFS [31], which Druid refers to as "deep storage". The ingest, persist, merge, and handoff steps are fluid; there is no data loss during any of the processes.

To better understand the flow of data through a real-time node, consider the following example. First, we start a real-time node at 13:37. The node will only accept events for the current hour or the next hour. When the node begins ingesting events, it will announce that it is serving a segment of data for a time window from 13:00 to 14:00. Every 10 minutes (the persist period is configurable), the node will flush and persist its in-memory buffer to disk. Near the end of the hour, the node will likely see events with timestamps from 14:00 to 15:00. When this occurs, the node prepares to serve data for the next hour and creates a new in-memory index. The node then announces that it is also serving a segment for data from 14:00 to 15:00. The node does not immediately merge the indexes it persisted from 13:00 to 14:00, instead it waits for a configurable window period for straggling events from 13:00 to 14:00 to come in. Having a window period minimizes the risk of data loss from delays in event delivery. At the end of the window period, the real-time node merges all persisted indexes from 13:00 to 14:00 into a single immutable segment and hands the segment off. Once this segment is loaded and queryable somewhere else in the Druid cluster, the real-time node flushes all information about the data it collected for 13:00 to 14:00 and unannounces it is serving this data. This process is shown in Figure 3.

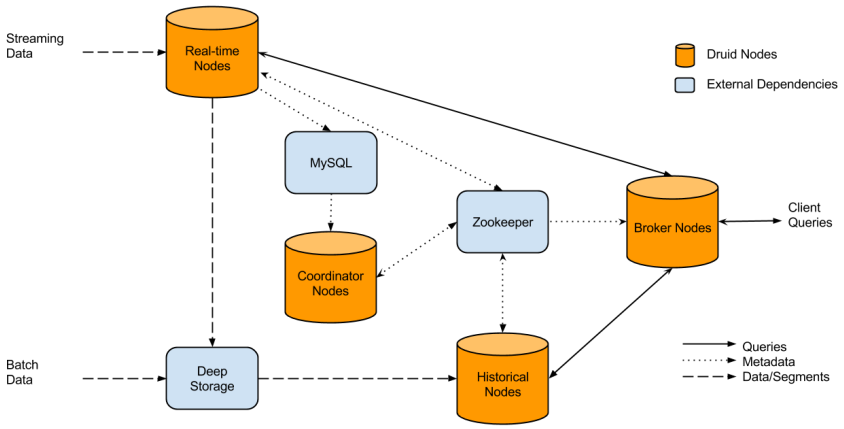

Figure 1: An overview of a Druid cluster and the flow of data through the cluster.

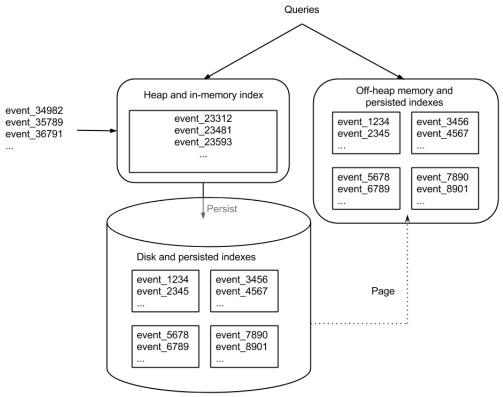

Figure 2: Real-time nodes first buffer events in memory. After some period of time, in-memory indexes are persisted to disk. After another period of time, all persisted indexes are merged together and handed off. Queries on data hit the in-memory index and the persisted indexes.

3.1.1 Availability and Scalability

Real-time nodes are a consumer of data and require a corresponding producer to provide the data stream. Commonly, for data durability purposes, a message bus such as Kafka [21] sits between the producer and the real-time node as shown in Figure 4. Real-time nodes ingest data by reading events from the message bus. The time from event creation to event consumption is ordinarily on the order of hundreds of milliseconds.

The purpose of the message bus in Figure 4 is two-fold. First, the message bus acts as a buffer for incoming events. A message bus such as Kafka maintains offsets indicating the position in an event stream that a consumer (a real-time node) has read up to and consumers can programmatically update these offsets. Typically, real-time nodes update this offset each time they persist their in-memory buffers to disk. In a fail and recover scenario, if a node has not lost disk, it can reload all persisted indexes from disk and continue reading events from the last offset it committed. Ingesting events from a recently committed offset greatly reduces a node's recovery time. In practice, we see real-time nodes recover from such failure scenarios in an order of seconds.

The second purpose of the message bus is to act as a single endpoint from which multiple real-time nodes can read events. Multiple real-time nodes can ingest the same set of events from the bus, thus creating a replication of events. In a scenario where a node completely fails and does not recover, replicated streams ensure that no data is lost. A single ingestion endpoint also allows for data streams for be partitioned such that multiple real-time nodes each ingest a portion of a stream. This allows additional real-time nodes to be seamlessly added. In practice, this model has allowed one of the largest production Druid clusters to be able to consume raw data at approximately 500 MB/s (150,000 events/s or 2 TB/hour).

3.2 Historical Nodes

Historical nodes encapsulate the functionality to load and serve the immutable blocks of data (segments) created by real-time nodes. In many real-world workflows, most of the data loaded in a Druid cluster is immutable and hence, historical nodes are typically the main workers of a Druid cluster. Historical nodes follow a shared-nothing architecture and there is no single point of contention among the nodes. The nodes have no knowledge of one another and are operationally simple; they only know how to load, drop, and serve

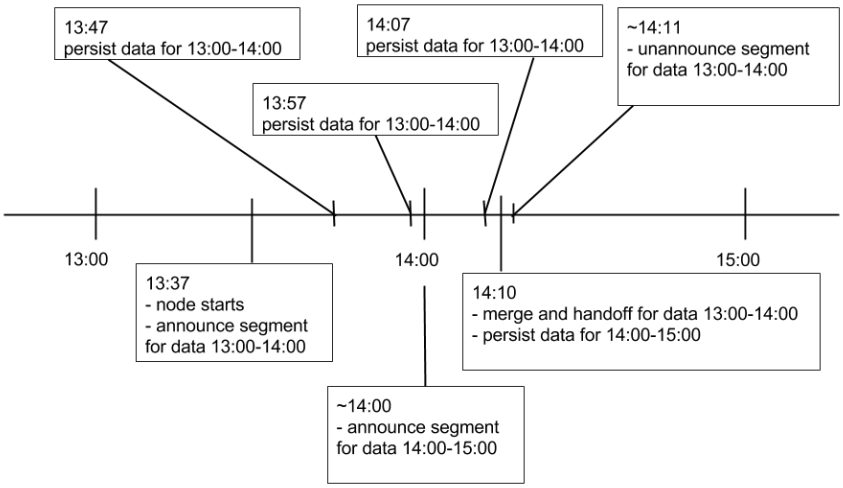

Figure 3: The node starts, ingests data, persists, and periodically hands data off. This process repeats indefinitely. The time intervals between different real-time node operations are configurable.

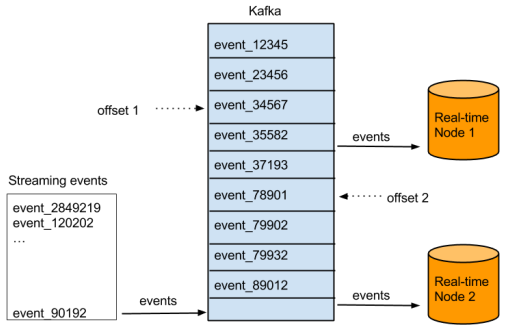

Figure 4: Multiple real-time nodes can read from the same message bus. Each node maintains its own offset.

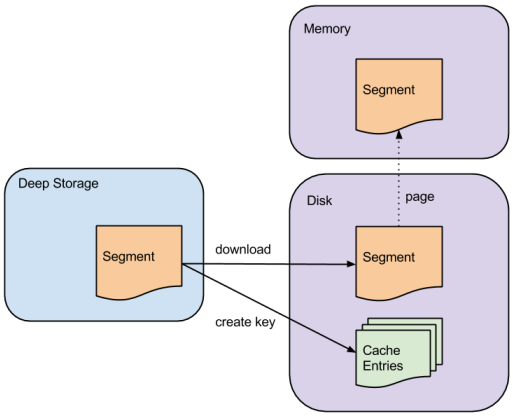

Figure 5: Historical nodes download immutable segments from deep storage.

immutable segments.

Similar to real-time nodes, historical nodes announce their online state and the data they are serving in Zookeeper. Instructions to load and drop segments are sent over Zookeeper and contain information about where the segment is located in deep storage and how to decompress and process the segment. Before a historical node downloads a particular segment from deep storage, it first checks a local cache that maintains information about what segments already exist on the node. If information about a segment is not present in the cache, the historical node will proceed to download the segment from deep storage. This process is shown in Figure 5. Once processing is complete, the segment is announced in Zookeeper. At this point, the segment is queryable. The local cache also allows for historical nodes to be quickly updated and restarted. On startup, the node examines its cache and immediately serves whatever data it finds.

Historical nodes can support read consistency because they only deal with immutable data. Immutable data blocks also enable a simple parallelization model: historical nodes can scan and aggregate immutable blocks concurrently without blocking.

3.2.1 Tiers

Historical 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 historical nodes that have a high number of cores and large memory capacity. The "hot" cluster can be configured to download more frequently accessed data. A parallel "cold" cluster can also be created with much less powerful backing hardware. The "cold" cluster would only contain less frequently accessed segments.

3.2.2 Availability

Historical nodes depend on Zookeeper for segment load and unload instructions. If Zookeeper becomes unavailable, historical nodes are no longer able to serve new data and drop outdated data, however, because the queries are served over HTTP, historical nodes are

still be able to respond to query requests for the data they are currently serving. This means that Zookeeper outages do not impact current data availability on historical nodes.

3.3 Broker Nodes

Broker nodes act as query routers to historical and real-time nodes. Broker nodes understand the metadata published in Zookeeper about what segments are queryable and where those segments are located. Broker nodes route incoming queries such that the queries hit the right historical or real-time nodes. Broker nodes also merge partial results from historical and real-time nodes before returning a final consolidated result to the caller.

3.3.1 Caching

Broker nodes contain a cache with a LRU [29, 20] invalidation strategy. The cache can use local heap memory or an external distributed key/value store such as memcached [16]. Each time a broker node receives a query, it first maps the query to a set of segments. Results for certain segments may already exist in the cache and there is no need to recompute them. For any results that do not exist in the cache, the broker node will forward the query to the correct historical and real-time nodes. Once historical nodes return their results, the broker will cache these results on a per segment basis for future use. This process is illustrated in Figure 6. Real-time data is 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.

The cache also acts as an additional level of data durability. In the event that all historical nodes fail, it is still possible to query results if those results already exist in the cache.

3.3.2 Availability

In the event of a total Zookeeper outage, data is still queryable. If broker nodes are unable to communicate to Zookeeper, they use their last known view of the cluster and continue to forward queries to real-time and historical nodes. Broker nodes make the assumption that the structure of the cluster is the same as it was before the outage. In practice, this availability model has allowed our Druid cluster to continue serving queries for several hours while we diagnosed Zookeeper outages.

3.4 Coordinator Nodes

Druid coordinator nodes are primarily in charge of data management and distribution on historical nodes. The coordinator nodes tell historical nodes to load new data, drop outdated data, replicate data, and move data to load balance. Druid uses a multi-version concurrency control swapping protocol for managing immutable segments in order to maintain stable views. If any immutable segment contains data that is wholly obseleted by newer segments, the outdated segment is dropped from the cluster. Coordinator nodes undergo a leader-election process that determines a single node that runs the coordinator functionality. The remaining coordinator nodes act as redundant backups.

A coordinator node 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, coordinator nodes maintain a Zookeeper connection for current cluster information. Coordinator nodes also maintain 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 table that contains a list of all segments that should be served by historical nodes. This table can be updated by any service that creates segments, for example, real-time nodes. The MySQL database also contains a rule table that governs how segments are created, destroyed, and replicated in the cluster.

3.4.1 Rules

Rules govern how historical segments are loaded and dropped from the cluster. Rules indicate how segments should be assigned to different historical 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. Rules are usually set for a period of time. For example, a user may use rules to load the most recent one month's worth of segments into a "hot" cluster, the most recent one year's worth of segments into a "cold" cluster, and drop any segments that are older.

The coordinator nodes load 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 may be configured. The coordinator node will cycle through all available segments and match each segment with the first rule that applies to it.

3.4.2 Load Balancing

In a typical production environment, queries often hit dozens or even hundreds of segments. Since each historical node has limited resources, 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 segments 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 historical 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 and may be discussed in future literature.

3.4.3 Replication

Coordinator nodes may tell different historical nodes to load copies of the same segment. 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 replicas. Replicated segments are treated the same as the originals and follow the same load distribution algorithm. By replicating segments, single historical node failures are transparent in the Druid cluster. We use this property for software upgrades. We can seamlessly take a historical node offline, update it, bring it back up, and repeat the process for every historical node in a cluster. Over the last two years, we have never taken downtime in our Druid cluster for software upgrades.

3.4.4 Availability

Druid coordinator nodes have two external dependencies: Zookeeper and MySQL. Coordinator nodes rely on Zookeeper to determine what historical nodes already exist in the cluster. If Zookeeper becomes unavailable, the coordinator will no longer be able to send

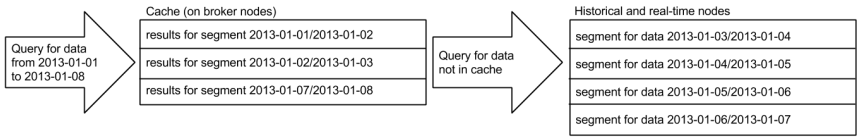

Figure 6: Broker nodes cache per segment results. Every Druid query is mapped to a set of segments. Queries often combine cached segment results with those that need tobe computed on historical and real-time nodes.

instructions to assign, balance, and drop segments. However, these operations do not affect data availability at all.

The design principle for responding to MySQL and Zookeeper failures is the same: if an external dependency responsible for coordination fails, the cluster maintains the status quo. Druid uses MySQL to store operational management information and segment metadata information about what segments should exist in the cluster. If MySQL goes down, this information becomes unavailable to coordinator nodes. However, this does not mean data itself is unavailable. If coordinator nodes cannot communicate to MySQL, they will cease to assign new segments and drop outdated ones. Broker, historical and real-time nodes are still queryable during MySQL outages.

4. STORAGE FORMAT

Data tables in Druid (called *data sources*) are collections of timestamped events and partitioned into a set of segments, where each segment is typically 5–10 million rows. Formally, we define a segment as a collection of rows of data that span some period in time. Segments represent the fundamental storage unit in Druid and replication and distribution are done at a segment level.

Druid always requires a timestamp column as a method of simplifying data distribution policies, data retention policies, and first-level query pruning. Druid partitions its data sources into well-defined time intervals, typically an hour or a day, and may further partition on values from other columns to achieve the desired segment size. For example, partitioning the data in Table 1 by hour results in two segments for 2011-01-01, and partitioning the data by day results in a single segment. The time granularity to partition segments is a function of data volume and time range. A data set with timestamps spread over a year is better partitioned by day, and a data set with timestamps spread over a day is better partitioned by hour.

Segments are uniquely identified by a data source identifer, the time interval of the data, and a version string that increases whenever a new segment is created. The version string indicates the freshness of segment data; segments with later versions have newer views of data (over some time range) than segments with older versions. This segment metadata is used by the system for concurrency control; read operations always access data in a particular time range from the segments with the latest version identifiers for that time range.

Druid segments are stored in a column orientation. Given that Druid is best used for aggregating event streams (all data going into Druid must have a timestamp), the advantages storing aggregate information as columns rather than rows are well documented [1]. 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 signficant performance degradations [1].

Druid has multiple column types to represent various data 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 page, user, gender, and city columns only contain strings. Storing strings directly is unnecessarily costly and string columns can be dictionary encoded instead. Dictionary encoding is a common method to compress data and has been used in other data stores such as PowerDrill [17]. In the example in Table 1, we can map each publisher to an unique integer identifier.

```
Justin Bieber   -> 0
Ke$ha -> 1
```

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

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

The resulting integer array lends itself very well to compression methods. Generic compression algorithms on top of encodings are extremely common in column-stores. Druid uses the LZF [23] compression algorithm.

Similar compression methods can be applied to numeric columns. For example, the characters added and characters removed columns in Table 1 can also be expressed as individual arrays.

```
Characters Added  -> [1800, 2912, 1953, 3194]
Characters Removed -> [25, 42, 17, 170]
```

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

4.1 Indices for Filtering Data

In most real world OLAP workflows, queries are issued for the aggregated results for some set of metrics where some set of dimension specifications are met. An example query may ask ”How many Wikipedia edits were done by users in San Francisco who are also male?”. This query is filtering the Wikipedia data set in Table 1 based on a Boolean expression of dimension values. In many real world data sets, dimension columns contain strings and metric columns contain numeric values. Druid creates additional lookup indices for string columns such that only those rows that pertain to a particular query filter are ever scanned.

Let us consider the page column in Table 1. For each unique page in Table 1, we can form some representation indicating in which

Figure 7: Integer array size versus Concise set size.

table rows a particular page is seen. We can store this information in a binary array where the array indices represent our rows. If a particular page is seen in a certain row, that array index is marked as `1`. For example:

```
Justin Bieber   -> rows [0, 1] -> [1][1][0][0]
Ke$ha -> rows [2, 3] -> [0][0][1][1]
```

`Justin Bieber` is seen in rows `0` and `1`. This mapping of column values to row indices forms an inverted index [34]. To know which rows contain `Justin Bieber` or `Ke$ha`, we can `OR` together the two 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. Popular algorithms include Byte-aligned Bitmap Code [3], Word-Aligned Hybrid (WAH) code [39], and Partitioned Word-Aligned Hybrid (PWAH) compression [37]. Druid opted to use the Concise algorithm [10] as it can outperform WAH by reducing the size of the compressed bitmaps by up to 50%. Figure 7 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 [36] 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 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 require less space than Concise sets and become a better alternative.

4.2 Storage Engine

Druid's persistence components allows for different storage engines to be plugged in, similar to Dynamo [12]. These storage engines may store data in an entirely in-memory structure 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. By default, a memory-mapped storage engine is used.

When using a memory-mapped storage engine, Druid relies on the operating system to page segments in and out of memory. Given that segments can only be scanned if they are loaded in memory, a memory-mapped storage engine allows recent segments to retain in memory whereas segments that are never queried are paged out. The main drawback with using the memory-mapped storage engine is when a query requires more segments to be paged into memory than a given node has capacity for. In this case, query performance will suffer from the cost of paging segments in and out of memory.

5. QUERY API

Druid has its own query language and accepts queries as POST requests. Broker, historical, and real-time nodes all 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, time range of interest, the type of request, and the metrics to aggregate over. The result will also be a JSON object containing the aggregated metrics over the time period.

Most query types will also support a filter set. A filter set is a Boolean expression of dimension name and value pairs. Any number and combination of dimensions and values may be specified. When a filter set is provided, only the subset of the data that pertains to the filter set will be scanned. The ability to handle complex nested filter sets is what enables Druid to drill into data at any depth.

The exact query syntax depends on the query type and the information requested. A sample count query over a week of data is shown below:

```
{
  "queryType"     : "timeseries",
  "dataSource"    : "wikipedia",
  "intervals"     : "2013-01-01/2013-01-08",
  "filter"        : {
    "type"        : "selector",
    "dimension"   : "page",
    "value"       : "Ke$ha"
  },
  "granularity"   : "day",
  "aggregations"  : [ {
      "type"      : "count",
      "name"      : "rows"
   } ]
}
```

The query shown above will return a count of the number of rows in the Wikipedia datasource from 2013-01-01 to 2013-01-08, filtered for only those rows where the value of the "page" dimension is equal to "Ke$ha". The results will be bucketed by day and will be a JSON array of the following form:

```
[ {
    "timestamp": "2012-01-01T00:00:00.000Z",
    "result": {
      "rows": 393298
    }
  },
  {
    "timestamp": "2012-01-02T00:00:00.000Z",
    "result": {
      "rows": 382932
    }
  },
  ...
  {
    "timestamp": "2012-01-07T00:00:00.000Z",
    "result": {
      "rows": 1337
    }
} ]
```

Druid supports many types of aggregations including double sums, long sums, minimums, maximums, and several others. Druid also supports complex aggregations such as cardinality estimation and approxmiate quantile estimation. The results of aggregations can be combined in mathematical expressions to form other aggregations. The query API is highly customizable and can be extended to filter and group results based on almost any arbitrary condition. It is beyond the scope of this paper to fully describe the query API but more information can be found online[3]. We are also in the process of extending the Druid API to understand SQL.

6. PERFORMANCE BENCHMARKS

To illustrate Druid's performance, we conducted a series of experiments that focused on measuring Druid's query and data ingestion capabilities.

6.1 Query Performance

To benchmark Druid query 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 and then on dimension values, creating thousands of shards roughly 8 million fact rows apiece.

The cluster used in the benchmark consisted of 100 historical 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, sufficiently fast Ethernet and more than enough disk space.

SQL statements are included in Table 2. These queries are meant to represent some common queries that are made against Druid for typical data analysis workflows. Although Druid has its own query

[3]http://druid.io/docs/latest/Querying.html

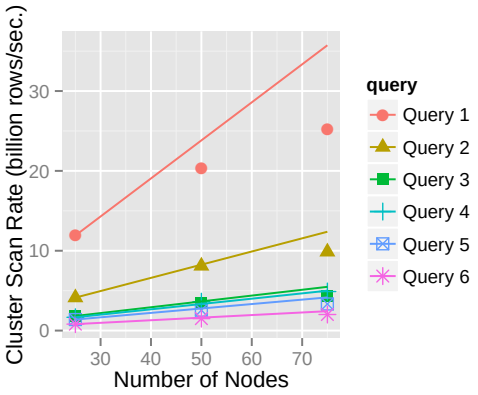

Figure 8: Druid cluster scan rate with lines indicating linear scaling from 25 nodes.

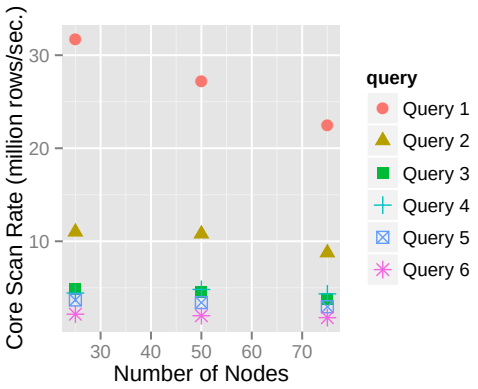

Figure 9: Druid core scan rate.

language, we choose to translate the queries into SQL to better describe what the queries are doing. 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 8 shows the cluster scan rate and Figure 9 shows the core scan rate. In Figure 8 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, the first SQL count query (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. However, queries 2-6 maintain a near-linear speedup up to 50 nodes: the core scan rates in Figure 9 remain nearly constant. The increase in speed 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 [2].

Table 2: Druid Queries

Query #	Query
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

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 historical 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.

6.2 Data Ingestion Performance

To measure Druid's data latency latency, we spun up a single real-time node with the following configurations:

- JVM arguments: -Xmx2g -Duser.timezone=UTC -Dfile.encoding=UTF-8 -XX:+HeapDumpOnOutOfMemoryError
- CPU: 2.3 GHz Intel Core i7

Druid's data ingestion latency is heavily dependent on the complexity of the data set being ingested. The data complexity is determined by the number of dimensions in each event, the number of metrics in each event, and the types of aggregations we want to perform on those metrics. With the most basic data set (one that only has a timestamp column), our setup can ingest data at a rate of 800k events/sec/node, which is really just a measurement of how fast we can deserialize events. Real world data sets are never this simple. To simulate real-world ingestion rates, we created a data set with 5 dimensions and a single metric. 4 out of the 5 dimensions have a cardinality less than 100, and we varied the cardinality of the final dimension. The results of varying the cardinality of a dimension is shown in Figure 10.

In Figure 11, we instead vary the number of dimensions in our data set. Each dimension has a cardinality less than 100. We can see a similar decline in ingestion throughput as the number of dimensions increases.

Finally, keeping our number of dimensions constant at 5, with four dimensions having a cardinality in the 0-100 range and the final dimension having a cardinality of 10,000, we can see a similar decline

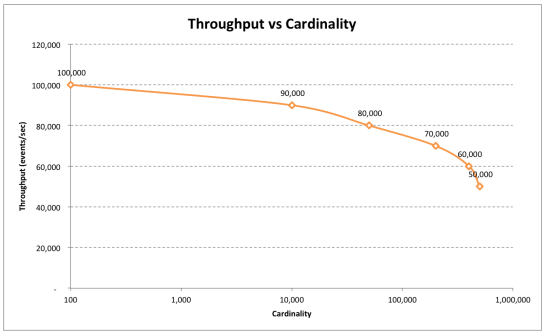

Figure 10: When we vary the cardinality of a single dimension, we can see monotonically decreasing throughput.

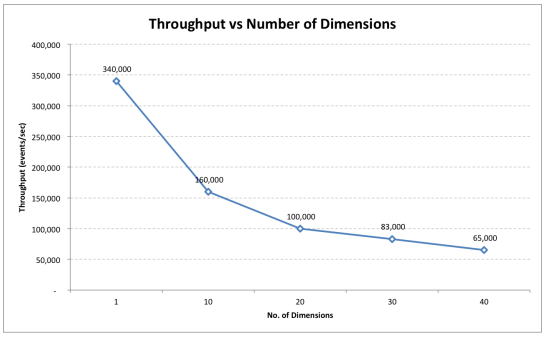

Figure 11: Increasing the number of dimensions of our data set also leads to a decline in throughput.

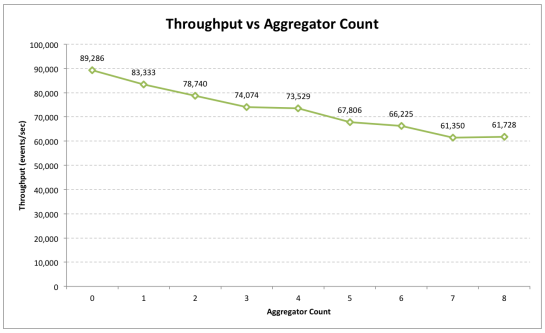

Figure 12: Adding new metrics to a data set decreases ingestion latency. In most real world data sets, the number of metrics in a data set tends to be lower than the number of dimensions.

in throughput when we increase the number of metrics/aggregators in the data set. We used random types of metrics/aggregators in this experiment, and they vary from longs, doubles, and other more complex types. The randomization introduces more noise in the results, leading to a graph that is not strictly decreasing. These results are shown in Figure 12. For most real world data sets, the number of metrics tends to be less than the number of dimensions. Hence, we can see that introducing a few new metrics does not impact the ingestion latency as severely as in the other graphs.

7. RELATED WORK

Cattell [7] maintains a great summary about existing Scalable SQL and NoSQL data stores. Hu [18] contributed another great summary for streaming databases. Druid feature-wise sits somewhere between Google's Dremel [27] and PowerDrill [17]. 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 columnar data stores [15], many of these data stores are designed to be more generic key-value stores [33] and do not support computation directly in the storage layer. There are also other data stores designed for some of the same of the data warehousing issues that Druid is meant to solve. These systems include include in-memory databases such as SAP's HANA [14] and VoltDB [38]. These data stores lack Druid's low latency ingestion characteristics. Similar to [30], Druid has analytical features built in, however, it is much easier to do system wide rolling software updates in Druid (with no downtime).

Druid's low latency data ingestion features share some similarities with Trident/Storm [26] and Streaming Spark [40], however, both systems are focused on stream processing whereas Druid is focused on ingestion and aggregation. Stream processors are great complements to Druid as a means of pre-processing the data before the data enters Druid.

There are a class of systems that specialize in queries on top of cluster computing frameworks. Shark [13] is such a system for queries on top of Spark, and Cloudera's Impala [9] is another system focused on optimizing query performance on top of HDFS. Druid historical nodes download data locally and only work with native Druid indexes. We believe this setup allows for faster query latencies.

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 optimization techniques such as using inverted indices to perform fast filters are also used in other data stores [25].

8. CONCLUSIONS

In this paper, we presented Druid, a distributed, column-oriented, real-time analytical data store. Druid is designed to power high performance applications and is optimized for low query latencies. Druid supports streaming data ingestion and is fault-tolerant. We discussed how Druid performance was able to scan 27 billion rows in a second. We summarized key architecture aspects such as the storage format, query language, and general execution. In the future, we plan to cover the different algorithms we've developed for Druid and how other systems may plug into Druid in greater detail.

9. ACKNOWLEDGEMENTS

Druid could not have been built without the help of many great engineers at Metamarkets and in the community. We want to thank everyone that has contributed to the Druid codebase for their invaluable support.

10. REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980. ACM, 2008.

[2] 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.

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

[4] Apache. Apache solr. `http://lucene.apache.org/solr/`, February 2013.

[5] S. Banon. Elasticsearch. `http://www.elasticseach.com/`, July 2013.

[6] 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.

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

[8] 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.

[9] Cloudera impala. `http://blog.cloudera.com/blog`, March 2013.

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

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] B. Hu. Stream database survey. 2011.

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

[20] 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.

[21] 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.

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

[23] Liblzf. `http://freecode.com/projects/liblzf`, March 2013.

[24] LinkedIn. Senseidb. `http://www.senseidb.com/`, July 2013.

[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. `http://storm-project.net/`, 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] Paraccel analytic database. `http://www.paraccel.com/resources/Datasheets/ParAccel-Core-Analytic-Database.pdf`, March 2013.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] E. Tschetter. Introducing druid: Real-time analytics at a billion rows per second. `http://metamarkets.com/2011/druid-part-i-real-time-analytics-at-a-billion-rows-per-sec` April 2011.

[36] Twitter public streams. `https://dev.twitter.com/docs/streaming-apis/streams/public`, March 2013.

[37] 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.

[38] L. VoltDB. Voltdb technical overview. `https://voltdb.com/`, 2010.

[39] 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.

[40] 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 Computing*, pages 10–10. USENIX Association, 2012.