Vortex: A Stream-oriented Storage Engine For Big Data Analytics

Pavan Edara
Google
Kirkland, WA, USA
epavan@google.com

Jonathan Forbes
Google
Kirkland, WA, USA
forbesj@google.com

Bigang Li
Google
Kirkland, WA, USA
bigli@google.com

ABSTRACT

Organizations are increasingly looking for ways to simplify collection and transformation of vast amounts of data collected from a highly connected internet. Data analytics over continuous streams of data enables interactive applications and reduces time to insights. Traditionally, streaming data collection and analysis has been either achieved by building systems, or using data warehouses built for batch processing. In this paper, we present Vortex, a storage engine that we built inside Google BigQuery to support real-time analytics. Vortex is a streaming-first storage system that supports both streaming and batch data analytics. Today, BigQuery uses Vortex to support petabyte scale data ingestion with sub-second data freshness and query latency.

CCS CONCEPTS

• **Information systems** → **Stream management**.

KEYWORDS

Streaming, Data Warehouse, Real-time, Analytics, Storage

ACM Reference Format:
Pavan Edara, Jonathan Forbes, and Bigang Li. 2024. Vortex: A Stream-oriented Storage Engine For Big Data Analytics. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3626246.3653396

1 INTRODUCTION

Unbounded data sources (eg. mobile devices, web click streams, telemetry from IoT devices) are creating massive amounts of data that needs to be stored and analyzed in real-time. Applications are advancing at a rapid pace to extract continuous insights from streaming data. Data warehouses and lakehouses provide managed data analysis services that are scalable, cost effective and easy to use.

These data warehouses often rely on storage systems such as distributed file systems or object stores to store massive amounts of data. Data warehouses typically store data in columnar storage formats [2] (either proprietary or open source). Tables are frequently partitioned and clustered by the values of one or more columns to provide locality of access for point or range lookups, aggregations, and updates of rows. Mutations (for example, SQL DML) modify individual rows across one or more blocks.

Streaming data brings unique challenges to analytics. To facilitate large scale analysis, applications that generate data typically push the data out to a remote distributed storage as quickly as possible. Some past systems have been architected to buffer data either locally or in a transient storage system. The data is then loaded into an analytic storage system to enable processing over it. While this works well for batch data, these systems are not entirely sufficient for streaming data because:

- The offline batch ingestion architecture sacrifices data freshness, a requirement fundamental to stream processing.
- Data generating applications sometimes run in highly resource constrained environments, where there is limited storage for buffering locally. To workaround the limitation, applications have to resort to techniques such as down-sampling to reduce data volume, as a result reducing data quality.
- Data is copied multiple times. Starting from the data source, it is first written to some temporary storage from which it is imported. This adds latency, impacts efficiency and creates challenges with data governance.

Contributions of this paper: We present Vortex, a storage engine that we built with streaming data as a first class concern. Instead of trying to adapt infrastructure built for batch data to work with streaming, we observe that it is better to build the storage system for streaming and then use it for batch. Vortex provides a highly distributed, regionally replicated storage engine that is optimized for append-focused ingestion of structured and semi-structured data. Vortex has been running in production at scale to support BigQuery. Vortex has the following key properties:

- **Consistent**: Guarantees ACID properties for all API operations.
- **Unified API for batch and streaming**: Vortex offers a single unified API with support for both streaming and batch data.
- **Scalable**: Vortex implements a fully distributed data and control plane and as a result supports tables of multiple petabytes size.
- **Performant**: The Vortex API offers sub-second tail write latencies that simplify client side application programming.

The rest of the paper is organized as follows. We present a brief survey of related work in Section 2. We provide a background of the architecture of BigQuery in Section 3. The API and design of Vortex's storage management system are presented in Section 4, Section 5 and Section 6 respectively. We then describe Vortex's integration with parallel data processing engines in Section 7. We show

SIGMOD-Companion '24, June 9–15, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0422-2/24/06
https://doi.org/10.1145/3626246.3653396

some real production results in Section 8. We present directions for future evolution of this work in Section 9.

2 RELATED WORK

Over the past few years, a number of data management systems have been built and continuously adapted to support data ingestion and processing for real-time applications such as log analytics. With the growing volume of data, the throughput requirements of these applications has increased exponentially. Additionally, we observe that applications have a broad spectrum of latency and freshness requirements on access to this data, and varying amounts of tolerance to fidelity of the results. Traditionally, different systems have been built to handle these different classes of applications. For instance, operational log analytics involves generating real time and historical insights on telemetry and other machine generated data. Specialized systems such as the Elastic stack [1] were created to address this segment. More generally, systems have either been tailored to batch or to streaming. Some systems have attempted to mimic the behavior of streaming by using a form of micro-batching [21] of data. Apache Kafka [11] is a distributed messaging system built for collection and delivery of data for real-time applications. Kafka's primary abstraction is a topic which is akin to a table in a database.

Pravega [9] organizes data into Streams, which are similar to topics in other messaging systems like Google Pub/Sub or Kafka. It supports arbitrarily large atomic transactions. Vortex is similar with its support for both coarse grained and fine grained transactions. Additionally, our system optimizes for large scale data analysis by continuously optimizing data and maintaining a combination of read and write optimized storage systems. Rockset [7] is a real-time analytics database that is similar to Vortex in that it offers both batch and streaming ingestion. Like Vortex, it implements a row store, column store, and uses an inverted index to support a variety of applications.

3 BACKGROUND

BigQuery is a fully-managed, serverless data warehouse that enables scalable analytics over petabytes of data. BigQuery architecture (shown in Figure 1) is fundamentally based on the principle of separation of storage and compute. BigQuery's storage engine is responsible for managing data. A horizontally scalable set of distributed compute nodes are responsible for data processing. These compute nodes can process data stored on variety of distributed storage systems. BigQuery decouples data shuffle from compute by building on top of disaggregated distributed memory. The Shuffle service facilitates communication between compute nodes. A set of horizontal services - APIs, metadata, security etc. realize the end-to-end functionality.

3.1 Query Execution Engine

Dremel [12, 13] is a distributed query execution engine that BigQuery uses to provide interactive latencies for analyzing petabyte scale datasets. BigQuery uses ANSI standard SQL as its query language API. BigQuery's data model has native support for semi-structured data [13]. Listing 1 shows a typical Sales table that takes advantage of repeated (ARRAY) and nested (STRUCT) fields and uses partitioning and clustering.

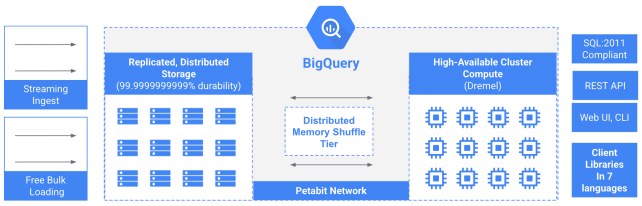

Figure 1: A high level architecture of BigQuery.

```
CREATE TABLE Sales(
  orderTimestamp TIMESTAMP,
  salesOrderKey STRING,
  customerKey STRING,
  salesOrderLines ARRAY<
    STRUCT<
      salesOrderLineKey INTEGER,
      dueDate DATE,
      shipDate DATE,
      quantity INTEGER,
      unitPrice NUMERIC>
  >,
  totalSale NUMERIC,
  currencyKey INTEGER)
  PARTITION BY DATE(orderTimestamp)
  CLUSTER BY customerKey
```

Listing 1: A typical table definition in BigQuery.

We now describe the major components involved in query processing. When a query is submitted, it is routed to one of the Query Coordinator nodes. As the name implies - the Query Coordinator is responsible for coordinating query execution. It parses the SQL query and algebrizes it to the Logical Query Plan. The query planner applies a number of logical plan transformations at that point, including pushing down computations and filters. The Query Coordinator then obtains a list of the tables involved in the query, columns requested from each table, and the filter predicates applied on top of table scans. The Query Coordinator uses this information in order to convert the Logical Query Plan to a distributed query execution plan.

The query plan can be described as a DAG (directed acyclic graph) of stages, where each stage is replicated across a number of workers which run the same set of operators over different pieces of data. The number of workers running for the given stage is the stage's degree of parallelism.

To execute the query, the leaf stages of the query plan are dispatched in parallel (subject to the amount of parallelism). The leaf stages of the query plan encapsulates a scan operation over the data source with any filters and computations that were pushed down. One such data source is BigQuery storage. Later in this paper, we discuss the details of the interactions with the Storage Engine.

3.2 Storage Engine

Dremel is a general purpose distributed query execution engine that can be used to perform in situ data analysis of semi-structured data. However, in order to address the problems of data management, mutating it with transactional consistency and providing rich governance features on it, we created the BigQuery storage engine. BigQuery storage provides a global namespace over all data in BigQuery. Data is organized into regional containers called datasets (analogous to schemata in traditional database management systems). Tables, logical views, materialized views, search indexes, stored procedures, machine learning models etc. all reside in a dataset. Users can access or mutate these objects using ANSI standard compliant SQL dialect. BigQuery offers APIs to bulk import data from object storage systems into its managed storage. The write API allows data ingestion and analysis at real time. The high throughput read API allows analysis of BigQuery tables from other data analytic engines like Google Cloud Dataflow and Apache Spark [20]. BigQuery's storage engine is built on top of Google's distributed file system called Colossus [10]. Data is replicated across multiple failure domains for disaster resilience. BigQuery storage supports ACID transactions with snapshot isolation. To provide this, BigQuery storage uses a metadata layer that stores metadata about user visible objects (Datasets, tables, view etc.) as well as system metadata about objects.

Figure 2 shows the architecture of BigQuery Storage. BigQuery's managed storage offering is a layer on top of BigQuery's Core Storage Engine. BigQuery tables can be partitioned and clustered by a set of columns in the table. BigQuery managed storage stores data in a proprietary columnar storage format called Capacitor [16]. BigLake Managed Tables (BLMTs) offer the fully managed experience of BigQuery tables while storing data in customer-owned cloud storage buckets using open file formats. Metadata management, data ingestion and data management operations on these tables are built on top of the same BigQuery core storage engine as BigQuery managed storage.

BigQuery uses Google's distributed, scalable and synchronously replicated database, Spanner to store coarse grained metadata. BigQuery also uses a columnar index on the metadata, called Big Metadata, for scale and accelerating query performance [8].

This paper focuses on Vortex, which is a key component of the BigQuery Core Storage Engine shown in Figure 2. Vortex is BigQuery's scalable, distributed and synchronously replicated storage engine that supports data ingestion, retrieval and curation. We will describe its api, architecture and its interaction with data processing both directly inside BigQuery as well as other data analytic engines like Google Cloud Dataflow.

4 STORAGE CONCEPTS AND API

BigQuery's data model supports tables with semi-structured data with repeated (ARRAY) and nested (STRUCT) columns. It provides a rich set of data types such as JSON, RANGE and BYTES, thus supported both structured and unstructured data. Vortex provides a stream abstraction on top of a BigQuery table.

4.1 Streams

A Vortex Stream is an entity to which rows can be appended to the current end. Each row in a Vortex Stream is identified by the Stream's identifier and its row offset within the Stream. Readers can concurrently read a Stream at different row offsets. A table is an unordered collection of Stream. A client process that wants to write data to a BigQuery table creates one or more Stream to write to it. Tens of thousands of clients can concurrently write to a table, each of them typically using their own dedicated Stream.

4.2 API

We present the key elements of Vortex API below using pseudo code.

4.2.1 Stream creation. Before writing data to a table, the Vortex client first creates a Stream on the table. Vortex supports three different types of Streams: UNBUFFERED, BUFFERED and PENDING. In an UNBUFFERED Stream, when an AppendStream request returns with success, it indicates that the input rows provided in the request have been durably committed to Vortex. Any subsequent reads of this table are guaranteed to see these rows. In a BUFFERED stream, the successful acknowledgement of an AppendStream request indicates that the input rows have been written to Vortex, but they aren't committed. These rows are not visible to subsequent reads until they are 'flushed'. In a PENDING Stream, rows are not visible until the Stream is committed.

```
enum StreamType {
  STREAM_TYPE_UNBUFFERED = 0,
  STREAM_TYPE_BUFFERED = 1,
  STREAM_TYPE_PENDING = 2,
};

CreateStreamOptions options;
options.set_stream_type(STREAM_TYPE_UNBUFFERED);
Stream s = CreateStream(table_name, options);
```

Listing 2: Creating a Vortex Stream

The CreateStream call in Listing 3 shows the creation of a Vortex Stream which returns a Stream object. The table schema is a property of this object.

4.2.2 Appending rows. Using the schema returned during Stream creation, the client serializes structured or semi-structured input data to a binary format. Vortex supports multiple data formats (such as Protocol buffers and Avro) and is extensible to other formats (e.g, Arrow).

```
RowSet row_set;
// Populate row_set from the input data and the schema.
// ...
// Append data to the end of a Stream.
AppendStreamResponse response = AppendStream(s, row_set,
      [row_offset]);
```

Listing 3: Appending to a Vortex Stream

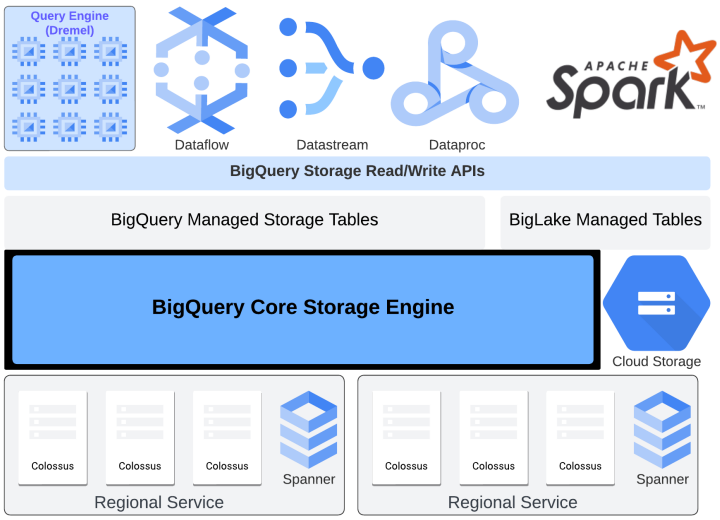

Figure 2: A high level architecture of BigQuery Storage.

The optional row_offset parameter allows the client to specify the offset in the Stream at which it expects the rows to be appended. As described above, a Stream has a single append point which is always at the current end of Stream. We call the number of rows written to a Stream its current length. Using an append offset that is different from the current length of a Stream will result in an error. Essentially, if multiple writes (from the same or different clients) attempt to use the same row_offset, only one of them will succeed, and all others will fail. This is useful also to ensure exactly-once semantics in the face of duplicate appends that use the same row_offset, which are a common technique for retries in a distributed environment. Clients can trade-off this idempotency guarantee in favor of reduced latency by omitting this optional parameter. In this case, the append is simply written to the current end of Stream, essentially providing at-least once semantics.

For performance and latency reasons, Vortex allows writes on a Stream to be pipelined. For example, consider a Stream that has 4 rows in it, i.e., it has a length of 4. The next available row_offset for append in this Stream is 4. If the first AppendStream call appended 10 rows starting at row_offset 4, a subsequent AppendStream call that wants to append the next 5 rows at row_offset 14 does not need to wait for the first AppendStream to finish. However, it is worth noting that the AppendStream calls must be issued in the order of row_offset. If the second call, with row offset 14, is received by the server before the first one (at row_offset 4), the offset validation logic on Vortex will fail the request because it expects the next append to use offset 4.

4.2.3 Flushing a stream. Recall that data written to a BUFFERED stream is not committed until it is flushed. Flushing is achieved by invoking the FlushStream API.

```
Status status = FlushStream(s, row_offset)
```

Listing 4: Flushing a BUFFERED Stream

If FlushStream returns success, it indicates that all rows in the Stream up-to and including the row at row_offset have been committed. If the current length of the Stream is less than row_offset, FlushStream returns an error. The FlushStream operation is idempotent, i.e., the same row offset can be flushed multiple times. The API allows the current length of the Stream, i.e. the number of rows appended to it so far, to be arbitrarily far ahead of the highest row offset up to which the Stream has been flushed.

4.2.4 Committing a stream. Data written to PENDING Streams are not visible until the Stream is marked committed. A common pattern with batch processing is for multiple workers to independently write to the table concurrently. In order to achieve atomicity of these parallel distributed writes, each worker creates a PENDING Stream. It writes all the data to it and reports its completion to a coordinator node. When the coordinator receives success from all the workers, it issues a batch commit request to Vortex to commit all the Streams atomically. This makes the data in all these Streams atomically visible to the readers.

```
std::vector<Stream> streams = GetStreams();
Status status = BatchCommitStreams(streams);
```

Listing 5: Committing PENDING Streams

4.2.5 Finalizing a stream. After a Stream is created, a client can write to the Stream forever. When the client has finished writing, it can finalize the Stream to prevent further appends to it.

```
Status status = FinalizeStream(s);
```

Listing 6: Finalizing Vortex Stream

4.2.6 Mutations. So far we have assumed that the row sets that are supplied to the AppendStream call are all insertions into the Stream. Vortex supports mutations using the AppendStream API. Before we describe the specification of our API for mutations, we note that BigQuery tables support the specification of unenforced primary key columns on the table. We call it unenforced because APIs that write data to a table do not enforce the uniqueness of the primary key.

Vortex defines a special virtual column called _CHANGE_TYPE in the table schema. This column can be specified for each row in the row set supplied to an AppendStream request. _CHANGE_TYPE indicates the type of the row being ingested into the Stream. It takes three values : INSERT, UPSERT and DELETE. The change type INSERT, which is the default, indicates that row must be appended to the table. UPSERT indicates intent to either update an existing row for the value of the primary key column(s) specified in the row or insert the new row otherwise. DELETE indicates that all rows with the primary key matching the value specified in the input row must be deleted. Absence of any matching row will return success. When a user uses only the UPSERT and DELETE change types, uniqueness of primary keys is enforced by construction.

The Vortex write API is available externally as the BigQuery Storage Write API [19]. It uses gRPC transport for high throughput transfer of data in binary format. A user stream could grow infinitely without finalize it.

5 ARCHITECTURE

5.1 Metadata concepts

In Section 4 we introduced the abstraction of a Stream in Vortex. A Stream is an append-only entity that acts as a conduit for writing data to a table. Internally, Streams are backed by the following entities.

Streamlets: Vortex Streams provide durable storage of data. A BigQuery region consists of 2 or more Borg clusters [18]. Each append to a Stream is durably written to 2 clusters before it is reported as success to the client. Data for a Stream can be in any 2 clusters of all the available clusters in a region. A Streamlet is a contiguous slice of rows in the Stream, all of which are present in the same 2 clusters. A Stream is an ordered list of one or more Streamlets. Given the Stream's append-only semantics, a Stream has at most one writable Streamlet. The writable Streamlet, if one exists, is always the last Streamlet in a Stream.

Fragments: Each Streamlet is further split into contiguous blocks of rows called Fragments. Fragments typically are a range of rows inside a log file. Log files are stored in Colossus.

Data formats: BigQuery operates broadly with data in two different classes of data formats. The write-optimized storage format (hereafter referred to as WOS) is the format in which data is written by Vortex's append API. The read-optimized storage format (hereafter referred to as ROS) is the format in which data is optimized for data processing. Typically, this is a columnar format. BigQuery Managed Storage Tables uses Capacitor as ROS, while BigLake Managed Tables use Parquet as ROS. In the rest of this paper, we'll use ROS as a generalization for these read-optimized formats.

Streamlets and Fragments are internal physical metadata entities; they aren't visible to the users of Vortex. Metadata for Streams and Streamlets is managed using a regional Spanner database.

Figure 3 shows the high level architecture of Vortex. The Vortex system contains the Control Plane which manages the metadata, the Data Plane which is responsible for writing, reading and management of the data, a Storage Optimization Service, and a thick client library that encapsulates the access paths to the control and data planes. The BigQuery engine directly uses the client library to access Vortex. Other processing engines use Vortex through the BigQuery Write API [19]. All services of Vortex run on Borg [18].

5.2 Control Plane

The Stream Metadata Server (SMS) is the control plane of Vortex. It manages the physical metadata of Streams, Streamlets and Fragments and is backed by a Spanner database which also stores the table's logical metadata. The table's logical metadata includes the table schema and user defined properties such as data partitioning and clustering.

Streams provide append points into a table. Multiple clients can append to the table and each client (typically) appends to its own Stream. A Stream typically contains a single Streamlet, but an additional Streamlet is created whenever a Streamlet is closed due to migrating the table to a new cluster, or due to a Stream Server restarting, load rebalancing, or an irrecoverable write error.

The SMS assigns a Streamlet to a specific Stream Server. The Stream Server maintains the set of fragments for the Streamlet.

When a Vortex client sends a request to append to a table, the SMS hands out a writable Stream handle which has not been assigned to another client. If there isn't one, it creates a new Stream, picks a Stream Server based on load and health characteristics and instructs it to create the Streamlet.

The SMS then responds to the client request with the Streamlet id and the address of the Stream Server with the writable Streamlet. The client creates a long-lived connection to the Stream Server to append batches of rows to the Streamlet.

5.2.1 Sharding the control plane. The Vortex SMS is deployed in multiple Borg clusters in a GCP region. BigQuery's multi-tenant architecture is made possible by an assignment algorithm that continuously determines the best placement of customer workloads in that region. Each customer workload is assigned a primary and secondary cluster [17]. When data is ingested into Vortex, each table is assigned to a primary cluster, and Vortex services in that cluster are responsible for managing data for that table. When a cluster is transiently unavailable or experiencing issues, Vortex transparently fails over the management of the table to the secondary cluster. As a result of the dynamic workload assignment process, Vortex in a region can be scaled simply by adding more clusters.

Within a single cluster, there are multiple SMS tasks. As described above, each table is assigned to a pair of primary and secondary clusters. Within the cluster, each active table's metadata is managed by a single SMS task. Assignment of table's to SMS tasks is done by Slicer [3] and is eventually consistent – this means that there can be rare times when two SMS tasks think that they both manage the table's metadata. Vortex is resilient to such inconsistency without affecting the correctness of the metadata. This is achieved by the

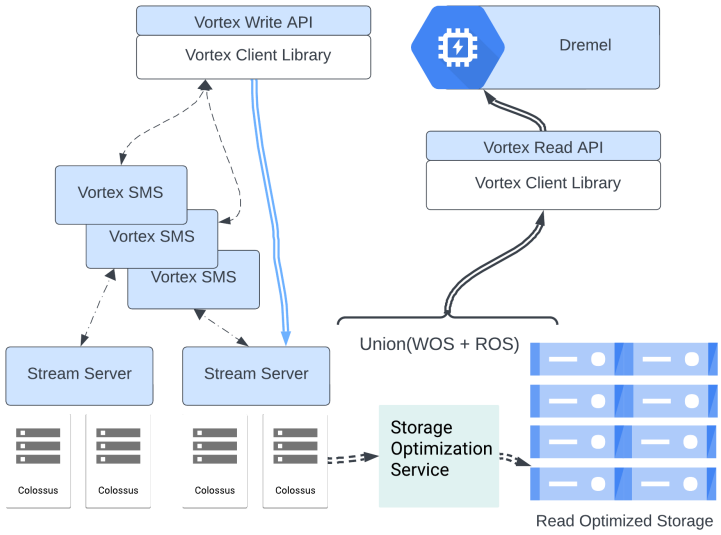

Figure 3: A high level architecture of Vortex.

ACID semantics offered by the Spanner transactions that are used to manage Stream and Streamlet metadata. When an SMS task "owning" a table becomes unavailable, Slicer redistributes the load by assigning the table to a new SMS task and notifying it. Load balancing of metadata operations across SMS tasks is achieved by reporting load information to Slicer, which then uses it to distribute load to other SMS tasks in the cluster.

5.3 Data Plane

The Stream Server is the data plane of Vortex. It owns a set of Streamlets and creates Fragments for those Streamlets. The Stream Server has its own in memory metadata about its Streamlets and Fragments, and persists this by writing to a transaction log and periodically writing checkpoints. After writing a checkpoint, old transaction logs and checkpoints are garbage collected. Fragments, checkpoints, and transaction logs are all stored in Colossus.

The Stream Server knows which Fragments belong to which Streamlet, their committed size, the minimum and maximum record timestamp in each Fragment, whether a Streamlet or Fragment is finalized, the schema version, and the partitioning and clustering columns of the table.

Each cluster often contains hundreds of Stream Servers. A given Stream Server in a cluster can host Streamlets for any table which uses the cluster as its primary. The SMS chooses the Stream Servers by balancing between CPU, memory and network traffic load.

When a Stream Server receives a request to create a Streamlet from the SMS, it persists this to its metadata and responds that the Streamlet is now ready to accept appends. A client can then start sending batches of append row data to the Stream Server.

The Stream Server appends row data to the latest Fragment in the Streamlet, and finalizes a Fragment and creates the next one when the current one reaches a certain size. The maximum size of a Fragment is chosen to be small enough that conversion by the Storage Optimization Service to the ROS format happens frequently, but not so small that too many Fragments are created in the metadata.

If an error is encountered while writing to a Fragment, the Stream Server finalizes the current Fragment and retries the append to the next Fragment. If subsequent retries fail, the Stream Server finalizes the Streamlet and fails the append request. The client will ask the SMS for a new Streamlet, which will most likely be created on a different Stream Server.

The Stream Server provides an API that returns the list of Fragments in a Streamlet, with the number of valid bytes to read from each Fragment.

5.4 Client Library

Vortex is accessed through a client library which supports reading from and writing to Vortex. It is a thick client library which can retry failed read and write operations. Failures can occur if a server does not respond quickly enough, or a table migrates to a new cluster and the current set of writable Streams is closed.

Frontend tasks that handle the BigQuery Storage Write API use the client library to write data to Vortex. The client library handles obtaining a writable Streamlet for a table from the SMS. If the table schema changes while a client is writing to it, the Stream Server fails the append and tells the client to obtain the latest schema. The client library obtains the new table schema from the SMS and then retries the append. Other write failures are handled by finalizing the current Streamlet, obtaining a new Streamlet from the SMS, and retrying the write there.

The query engine uses the read functionality of the client library, which handles reading Fragments and Streamlets. The client library is responsible for decrypting and decompressing data read from Fragments. The metadata associated with each Fragment lists all clusters that store a replica of that Fragment. The client always tries to read Fragments from the local cluster, but if this fails, it will automatically retry the read against a different cluster.

5.4.1 Schema Evolution. Table schema changes are effected on the SMS. Since the SMS does not have a connection to clients actively writing to that table, there is no direct way for it to notify the clients. Instead, when the table schema changes, the SMS first notifies the Stream Servers with currently writable Streamlets about the new schema version via heartbeat. The Stream Server then relays this information to the clients when they try to append. After the client learns about the new schema version, it fetches the updated schema from the SMS.

5.4.2 Unary and bi-directional RPC. We observe that only 10% of the Streams hold 90% of the data. The remaining 90% of the Streams are small in the volume of data they hold. To support applications with such a wide variety of write patterns, the Vortex client library can adaptively switch between using a single directional (unary) short-lived connection and a bi-directional long-lived connection.

The short-lived unary connection type supports a request-response mechanism, and optimistically re-uses connections using connection pooling. This connection type is efficient when append requests to a table are infrequent, as it does not incur the memory cost of holding open a persistent connection.

The long-lived bi-directional connection type supports streaming RPCs (such as sending a single large RPC as multiple smaller pieces), and allows multiple pipelined append requests. A client can continually append rows as they are received from the front end, before receiving the response from the previous append request. A persistent connection is very CPU efficient when processing a high volume of RPCs, but has a higher memory overhead due to its persistence and tracking multiple operations on the same connection. It is also more complicated to support, as the server can simultaneously process multiple incoming requests for the same Streamlet.

Bi-directional connections also support flow control. The Stream Server uses flow control to throttle incoming appends when there is a large amount of data in-flight. In-flight data remains in memory until it has been committed, but if Colossus is slow, flow control protects the Stream Server from running out of memory by limiting the rate of data it will accept. Flow control on a dedicated connection allows the stream server to limit the rate of data ingress it will accept on a per connection basis.

5.4.3 Lifetime of data in a Stream. When a client sends a request to the SMS to create a new Stream, the SMS generates a unique random id for the Stream, and the first Streamlet in that Stream, then attaches the stream to the table by persisting it into Spanner. The SMS sends a RPC to the Stream Server to create a Streamlet with the provided id. The SMS responds to the client that it can start appending to the Streamlet with this id on that Stream Server.

After this, the client directly streams data in batches of rows to the Stream Server. The Stream Server creates new log files as needed to store the row data. The only time the client ever goes back to the SMS is to perform reconciliation of failures such as when the Stream Server is unavailable or when the Stream Server deemed the Streamlet to have failed irrecoverably.

The Stream Server heartbeats every few seconds with SMS to inform it of changes to the Streamlets that it had been writing to.

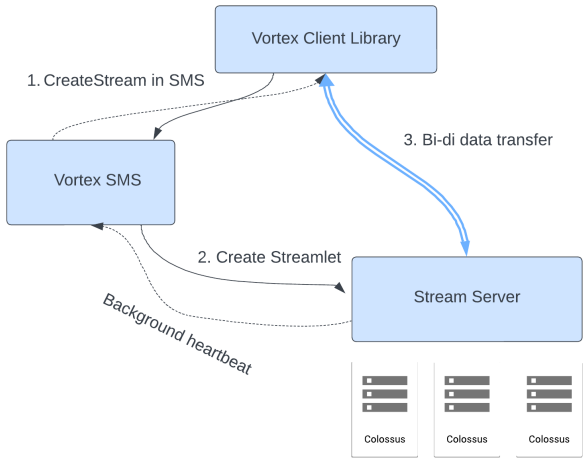

Figure 4: Life of a Vortex Append request

The Storage Optimization Service consults the SMS for a set of candidate Fragments, converts them to ROS, and commits them as live while marking the Fragments as deleted.

After a configurable period of time, the SMS tells Stream Servers via heartbeat to garbage collect the deleted Fragments. When the Stream Server acknowledges it has deleted the Fragments, the SMS deletes the Fragments from Spanner. Fragments that are deleted are kept sufficiently long to ensure that any active queries that are reading from them do not fail.

User initiated actions such as deletions of tables, datasets and projects can trigger garbage collection. As a catch all, a "groomer" job runs periodically to detect Fragments, Streams, or Streamlets that may be orphaned by these user initiated actions.

The typical heartbeat from Stream Server to SMS only contains metadata changes observed by the Stream Server since its previous heartbeat for the same Streamlet. To guard against accidental bugs, the Stream Server periodically sends a full state snapshot of all Streamlets it owns to the SMS. If the SMS responds that it is not aware of a Streamlet existing in any table, the Stream Server deletes the Streamlet. While performing these deletions, the system ensures that the Streamlet is sufficiently old. This avoids any in-flight races, and also provides an opportunity to investigate the reasons for its occurrence.

5.4.4 Fragment File Format. Each Fragment begins with a header which contains the File Map. The File Map lists the committed size and record ranges of all previous Fragments in the same Streamlet which have not yet been deleted. The File Map is used for disaster resilience.

The Stream Server buffers up to 2MB of records into a single write to a Fragment. Buffering 2MB enables better compression and avoids sending a large number of small writes to the file system. The write includes a header, which specifies a single server-assigned TrueTime [6] timestamp for all rows in the write. Use of TrueTime ensures that records are guaranteed to be assigned a timestamp with a bounded amount of clock skew in single digit milliseconds,

regardless of the Stream Server they were written to, and that a query is guaranteed to return data that was just written.

When a Fragment is finalized, the Stream Server appends a bloom filter, followed by a fixed length footer which describes the offset in the Fragment where the bloom filter starts. The bloom filter marks which key values are present for the partitioning and clustering columns.

The Fragment data format stores metadata records for FlushStream calls on BUFFERED streams. A flush operation is a metadata write to the Fragment which advances the committed row offset making all rows in the Streamlet (and the stream) up to that point visible.

5.4.5 Compression and Encryption. The Stream Server uses the Snappy compressor, which has a negligible CPU impact, to compress rows before appending them to the Fragment. This is more effective the larger the size of the batched append, and is highly effective on string data, which tends to be the majority of a row's size.

The typical compression ratio is 4:1 but can be 10:1 if values of string fields are common between many rows. Compression reduces not only the physical byte count residing in the WOS system, but also results in fewer bytes written and read over the network.

After compressing the data, the Stream Server encrypts the data before writing to Fragments, using either the system's encryption key or a customer supplied encryption key. Data is therefore in encrypted form while being sent over RPC to Colossus, while at rest, and while being read back.

Vortex uses an end-to-end CRC to protect row data as it is sent from the client to the Stream Server, and from the Stream Server to Colossus. The data bytes are sent alongside their CRC, and if one or the other is corrupted while in memory or in flight, Colossus will ultimately discover this and fail the write.

To protect against data being corrupted during compression, the Stream Server decompresses the compressed output and verifies the CRC was identical to that of the original uncompressed data.

5.4.6 Cross cluster/region read. The Vortex Client Library allows a reader in any region or cluster to read from any other. Due to network bandwidth limitations, reading cross region can be prohibitive. BigQuery always prefers to run compute in the region where the data is. Further, to reduce cross cluster network usage, BigQuery chooses the placement of compute while balancing flexibility and network transfer costs.

5.5 Heartbeat and Load balancing

The Stream Server sends a heartbeat to each SMS every few seconds to inform it about changes to Streamlet metadata as a result of new appends. These changes include creation of new log files, an increase in size of existing log files, and column properties of data in the log files. Along with per-Streamlet metadata, the Stream Server also sends its current load information (CPU, memory and append throughput) to the SMS. The Stream Server's quarantine status is also passed to the SMS for graceful handling of routine maintenance tasks such as rollouts and scaling up and down of the Stream Server task pool.

The SMS caches the Streamlet and Fragment level information in Spanner. This cache allows it to answer read requests at the time of query processing, making it possible to easily discover all the Fragments of that need to be read to answer queries to a table. The SMS aggregates the stream server's load information and uses it to load balance the assignment of new Streamlets to that Stream Server.

5.6 Disaster Resilience

The Stream Server performs synchronous replication by simultaneously writing fragments to two Colossus clusters before returning success to the client. Data in each Colossus cluster is already replicated to multiple servers and is resilient to the permanent failure of servers in a cluster. The replication described here is at a layer above the replication within a single cluster. It provides resilience against failures of entire clusters. We refer to the copy in each Colossus cluster as a replica. Vortex uses physical replication, meaning that the Stream Server log file writes are identical in both clusters. If either or both writes fail or time out, the current log file is closed, and the write is retried on the Stream Server to the new Fragment. The File Map in the header of the new Fragment specifies the committed final file size of the previous file with the failed writes. If the Stream Server is unable to continue writing to the Streamlet by using local retries, such as when a cluster is unavailable, it reports the failure to the client library. The client library informs the SMS and starts a reconciliation process. It determines the current length of the Streamlet by inspecting the metadata blocks in all replicas of the log files that are reachable. To poison writes from any zombie Stream Server processes that may still be writing to the log file, a sentinel record is written to the log files. This sentinel record invalidates the Stream Server's assumption that it is the sole writer to the log file, and causes it to relinquish ownership. After determining the reconciled length, the SMS records this information in Spanner. This algorithm ensures the data in the primary and secondary cluster are logically in sync even in the case of rare IO events.

6 DATA MANAGEMENT

In this section we'll describe the data and metadata management functionalities of Vortex.

6.1 Storage Optimization

A background service continuously optimizes data in Vortex as it is written. The goal of Storage Optimization is to optimize the format and layout of the data for large scale analysis. In doing so, it maintains an LSM[15] tree of Fragments, starting with Fragments in WOS at the deepest level of the tree, with progressively more optimized ROS versions as we climb up the tree.

The first step of optimization, shown in Figure 5, converts data that is in the WOS to ROS. As we mentioned before, ROS uses a columnar data format that is optimized for data analysis. To do this conversion, the optimizer first determines a list of unconverted Fragments as candidates from the SMS. It schedules workers to convert these Fragments to a smaller number of ROS Fragments.

To track the lifetime of Fragments, each Fragment maintains two timestamps: a **creation_timestamp** and a **deletion_timestamp**.

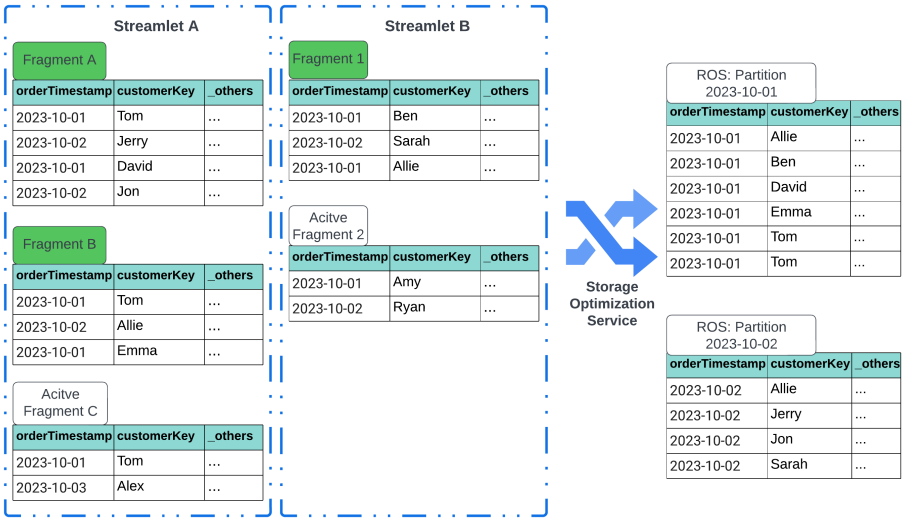

Figure 5: Conversion of Write Optimized (WOS) to Read Optimized Storage (ROS).

A Fragment is visible to requests that read the table at a snapshot timestamp that is within the interval [*creation_timestamp*, *deletion_timestamp*). At each step of the optimization, the optimizer atomically sets the deletion_timestamp for the previous version of the fragments and the creation_timestamp for the new version. This guarantees that a row is included exactly once when reading the data from storage.

Automatic Reclustering: BigQuery allows users to cluster the data in a table on a set of specified columns. Clustering defines a "weak" sort order on the data blocks in the table. In other words, BigQuery attempts to distribute the data such that the blocks store non-overlapping ranges of values for the clustering keys. Organizing data in non-overlapping ranges results in more efficient processing at read time, by improving partition pruning [14] and/or by reducing intermediate data transferred between query processing stages. BigQuery automatically determines the clustering key ranges of these new blocks as data is written. In steady state, most of the data is in non-overlapping ROS blocks, referred to as the baseline. The fraction of data that is in such non-overlapping ROS blocks is referred to as it's clustering ratio. As new data (referred to as the delta) is inserted into a table, the new blocks overlap with the existing blocks in the baseline. Once the delta is sufficiently large, the optimizer first range partitions the delta locally. After the deltas have accumulated sufficient data comparable in size to the size of the current baseline, the baselines and deltas are merged to generate a new baseline.

Figure 6 shows automatic reclustering of the baseline blocks A through X with the 2 delta blocks. The baseline blocks are non-overlapping in the values of the 'customerKey' clustering column. The result of the reclustering is the a new baseline that is non-overlapping in the clustering columns values and larger than the original baseline.

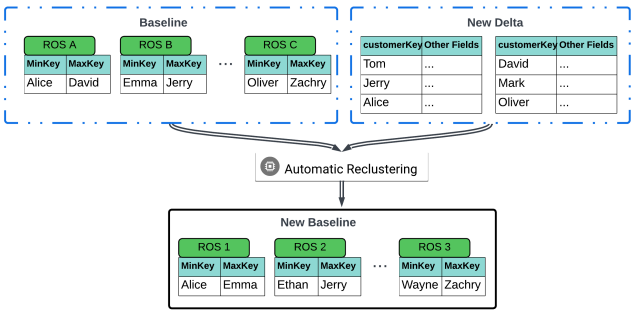

Figure 6: Automatic Reclustering.

6.2 Metadata Management

Vortex continuously tracks metadata for Streams, Streamlets, and Fragments. It maintains the coarse grained metadata such as Stream and Streamlets in a Spanner database. An example of coarse grained metadata is the state of a Streamlet that indicates whether the Streamlet is currently writable and its current length. The source of truth for Streamlet and Fragment metadata is persisted in the Vortex stream server's log. The Streamlet metadata in Spanner is for the most part a cache, and is not the source of the truth until the Streamlet is marked finalized. As the storage optimizer moves data between the layers in the LSM tree, BigQuery's highly scalable metadata management system, called Big Metadata [8] manages fine grained column properties for accelerating query performance.

In steady state, there is a tail of the Fragment and Streamlet metadata that may have not yet been indexed by Big Metadata. As the metadata of these blocks churns rapidly, we observe that scanning through the list of these tail blocks that need to be read to satisfy the snapshot read, can add latency to query processing. To address this, we continuously compact the metadata entries for

the Fragments by keeping the entries corresponding to the live fragments (i.e. fragments with deletion_timestamp unset) together in our log. This is done by maintaining a watermark which is the timestamp of the oldest live Fragment that has not yet been optimized. The compaction process is triggered by the optimization of Fragments by the Storage Optimizer.

6.3 Data Verification

Vortex continuously traces requests to detect data correctness issues such as missing or duplicated records. The system tracks all calls to the client library and saves aggregated information back to Vortex itself for large scale data analysis. Further, it also tracks storage optimization activity, which copies records between generations.

Using the above information, we have data verification pipelines to continuously validate the correctness of these operations. For every successful Vortex API call, we verify that the corresponding Stream, Streamlet and Fragment aree created in the system and that the appended data exists at the expected location (Stream + row_offset). We then verify that each append in the system reports a unique location. Finally, we also verify that each record is reported as converted exactly once from WOS to ROS. Additionally, for each conversion, we validate that the output records are consistent with the input records. These verification pipelines execute continuously as SQL queries in BigQuery.

6.4 BigLake Managed Tables

The Vortex storage engine also supports BigLake Managed Tables (BLMTs) which store data in the Apache Iceberg format in customer owned buckets in cloud storage. The WOS for these tables is in Colossus, in exactly the same way as BigQuery Managed Storage Tables. Metadata and data management relies on the same underlying service. The process of converting data from WOS to ROS writes Parquet files to the customer provided cloud storage buckets. Queries over BLMTs read the union of the WOS data and the Parquet files.

7 DATA ANALYTICS

Dremel is BigQuery's distributed SQL query processing engine. BigQuery tables can also be processed using Dataflow, or Dataproc. At a high level, to process a table, a processing engine requests the partitioned metadata for the table as of a specific snapshot read time. When Vortex SMS receives this request, it returns the union of the data in WOS and ROS. The WOS for a table consists of Streamlets and Fragments. As explained before, the SMS caches information about fragments in Spanner. This information is stale, as it receives heartbeats from a given Stream Server only every few seconds. A Streamlet could have additional data that is yet unknown to the SMS. In summary, the SMS returns the list of locations (in Colossus) for the unfinalized Streamlets and Fragments known to it. To fully read the WOS, the processing engine reads the Fragments and the portions of the unfinalized Streamlets that are not present in the list of Fragments. In the case of Dremel, the Query Coordinator receives this information and dispatches these Fragments and Streamlets to different Dremel shards to process them in parallel. Dremel shards read the data using the mechanism described in Subsection 7.1.

7.1 Reading data

Query processing in BigQuery reads data in Vortex directly from Colossus through a thick client library without contacting the Stream Server. If the Stream Server is available, it can serve its in-memory metadata for the client to determine the Fragments and offsets to serve the read. This is the common case, but it is purely an optimization. In the event of transient unavailability of the Stream Server, a client can find, on Colossus, the latest Fragment that contains data for that Streamlet. The File Map of the latest log file includes the committed final size of each of the previous Fragments. This serves as a replica of the information that would otherwise be available from the Stream Server. Clients will not read past the logical finalized size of a Fragment in the File Map, so will ignore failed or partial writes at the end of a Fragment.

The Stream Server only continues writing to a Fragment if the previous write succeeded to both replicas. In addition, the Stream Server performs a separate write of a "commit" record after each append. In the common case, i.e. an active Streamlet, this commit record is combined with the next data append. Otherwise, it is written after a small period of inactivity.

Given the above logic, if a reader sees that a Fragment contains any additional data after an append it just read, it knows that append is considered committed by the system. When reading the final append in the Fragment, it will typically see there is a commit record afterwards, and know the append was written to both replicas. If a reader encounters an append timestamp greater than the read snapshot timestamp, it can stop reading. This is common for an active table constantly receiving new records.

Reconciliation of the final append: When the client needs to read the final append in a Fragment, but the replicas have different sizes, or one replica is unavailable, it cannot make a local decision. A different client could see only one replica (and it could be a different replica) and come to a different conclusion about whether the append is committed. Vortex guarantees that it will return the same rows regardless of which replica is used for reading.

In these cases the client requests the SMS to reconcile the state of the final append. The SMS runs the reconciliation protocol described in Subsection 5.6 to determine the length of the Streamlet and the physical sizes of the Fragments. This allows the client to then decide whether the final append needs to be read or not.

7.2 Partition Elimination

Partition elimination (sometimes called partition pruning) [14] is a common technique to improve query performance. In partition elimination, the query coordinator inspects the filter condition and eliminates scan (and sometimes dispatch) of the partitions which cannot possibly satisfy the filter condition. In the context of Vortex, a partition refers to each of the Fragments and Streamlets returned by the SMS. Vortex performs partition elimination by maintaining column properties such as min/max values and bloom filters for columns on which the data is partitioned or clustered in BigQuery. These properties are maintained by the Stream Server for each Streamlet as data is written to it. Once a Fragment is marked finalized, these properties are communicated to the SMS, for caching. The properties for the tail of a Streamlet are maintained

by the Stream Server. As described in Big Metadata [8], when a query is received, BigQuery uses the filters specified in the query to construct derivative expressions on the column properties. The stored column properties are used to evaluate these expressions for each Fragment and Streamlet returned by the SMS to determine whether it is relevant to the query.

7.3 Mutations

BigQuery supports mutating SQL DML statements such as UPDATE, DELETE and MERGE. To enable this, Vortex allows a range of rows in a Fragment or Streamlet to be marked as deleted. A DELETE statement first determines the candidate rows to be marked deleted and at commit time persists a deletion mask to the Streamlet or Fragment metadata. To limit the size of these deletion masks, sometimes rows unaffected by the DML statement may also be marked deleted. We call these reinserted rows. In this case, copies of reinserted rows are written and committed to the table atomically along with the commit of the deletion mask. UPDATE statements are implemented as a combination of deletion of the old rows and an insertion of the updated rows.

The mechanism described above assumes that the SMS is already aware of the Fragments affected by the DML. Recall that the SMS may not be aware of some of the most recent Fragments in the Streamlet since they are asynchronously reported by the Stream Server via heartbeat. We call this unreported portion of the Streamlet its tail. When a DML statement needs to delete records in the Streamlet tail, the SMS marks the entire Streamlet tail as deleted, and similarly to the Fragments, the reinserted rows in the tail are copied over by the DML. The DML statement's commit is completed at this point. Subsequently, when the Stream Server's heartbeat informs the SMS about the Fragments in the tail, the SMS maps the deleted record range recorded in the Streamlet as part of DML commit, to deletion masks of the Fragment.

On a subsequent read following this deletion, the client library receives the deletion masks corresponding to each of the Streamlets and Fragments from the SMS. The reader applies the deletion mask to filter out the deleted rows.

As described in Subsection 6.1, during Storage Optimization, Vortex marks old Fragments as deleted and commits optimized Fragments. Marking Fragments as deleted can race with concurrently running DML operations which have already marked parts of a Fragment as deleted. Since Storage Optimization operates continuously, it would create conflicts for DML. We address this by yielding Storage Optimization to DML; whenever a DML statement is running, storage optimizer will not commit.

This introduces a problem when there is a single long running DML statement or a continuous stream of DML statements. In this scenario, the Optimizer might accumulate a large backlog of work, resulting in data not being optimized quickly for analysis in turn causing poor read performance on the table. To address this, Vortex supports a stable 1:1 conversion from a Fragment in WOS to a Fragment in a ROS. This prevents any issues due to the race, since the DML can still set the deletion masks on the read optimized Fragment, the same way it would have otherwise set on the original one in WOS.

7.4 Exactly-once Processing

The Vortex API described in Section 4 allows analytic engines such as BigQuery and Dataflow [5] to achieve end-to-end exactly once processing for both ingress and egress. This section will describe how we achieve this in the context of streaming Dataflow. Dataflow is based on the open source Apache Beam project. Beam provides a number of sinks to which data can be written. To output data to BigQuery, the user implements an application that looks similar to the code in Listing 7.

```
String tableName = getTableName();
PCollection<TableRow> tableRows = getTableRows();
tableRows.apply(BigQueryIO.writeTableRows())
    .to(tableName)
    .withWriteDisposition(WRITE_APPEND);
```

Listing 7: Sample Beam code that writes to a BigQuery Table

Dataflow guarantees exactly-once semantics for data processing through the pipeline. However, when data is output to a sink it creates a side-effect in an external system. Achieving end-to-end exactly once guarantees relies on that external system. The 'BigQueryIO.writeTableRows()' function in the Beam BigQuery sink is implemented using the Vortex API. To achieve exactly-once, the sink operates in two stages. The first stage, called the Append stage, receives a partitioned stream of rows from its input. Rows in this stream are deterministically partitioned and partitions of the key space (or groups thereof) are each handled by a single worker. In rare scenarios, a worker may enter a zombie state due to network partitions etc., leading to a single partition's key range being handled by multiple workers at the same time. Each worker in the Append stage creates its own dedicated BUFFERED stream on the table. Along with the Stream identifier, it also keeps track of the row offset of the next write. It reads the next batch of rows (called a bundle) from Shuffle [4] and writes to its dedicated Stream at the row offset. It advances the row offset in its state by the number of rows in the bundle. Recall that with a BUFFERED stream, rows are not committed when a successful response is received from AppendStream. A subsequent FlushStream call that includes all the rows up to the end row offset will mark them committed. The Beam sink will perform this FlushStream call in a separate stage, called the Flush stage. After each successful AppendStream call the worker:

- Marks the rows in the input bundle as processed.
- Writes the Stream's identifier and the row offset for the FlushStream call to shuffle. This will make them available to the Flush stage.
- Updates the Stream's new length and offset in a state store.

Dataflow guarantees that these three modifications are committed atomically. Rarely, zombie workers may process input rows that were already previously marked as processed. This guarantee ensures that such workers cannot commit the results of this processing. In the context of the Append stage, the results of the processing the same row may be appended multiple times to the same Vortex Stream (at different offsets), but only one worker will succeed in marking that row as processed. This will prevent the

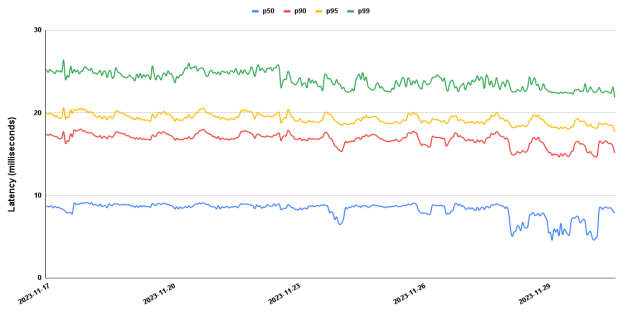

Figure 7: Vortex Append latency distribution

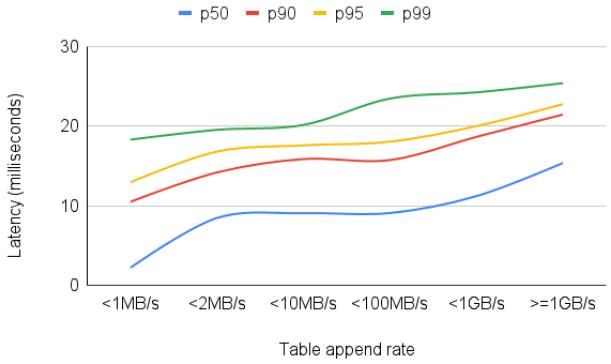

Figure 8: Vortex Append latency distribution by Append rate

stream identifier and row offset for FlushStream call from being written to Shuffle.

7.5 Unifying batch and streaming

Apache Beam [5] introduced a unified model for defining parallel data processing pipelines for streaming and batch analysis. The Vortex API brings that same unification to data ingestion into a data warehouse. With PENDING streams, Vortex guarantees ACID semantics on arbitrarily large transactions. Large transactions are common in batch ETL processes that do a periodic merge of data and require atomic commits of the changes. With BUFFERED Streams, as described in Subsection 7.4, Vortex provides Stream level micro-transactions with the same atomicity guarantees for smaller transactions. This API unification helps simplify application development without requiring the use of different systems for batch and stream processing.

8 RESULTS

Vortex serves BigQuery streaming traffic in production for petabyte scale datasets and supports throughput of multiple GB/sec over a given table. In this section we'll share some results that we see in our production environment.

Figure 7 shows the 50th, 90th, 95th and 99th percentile latency of Vortex Stream append requests over a 2 week period. As the graph shows, Vortex achieves very low latencies with a 50th percentile latency of 10 milliseconds and a 99th percentile of approximately 30 milliseconds. Furthermore, along with these low latencies, Vortex offers read-after-write consistency.

Figure 8 shows the latency distribution for tables grouped by their throughput. The tail latencies at the 99th percentile are labeled p99 in the graph. We see that across a wide range of throughput starting from tables with less than 1MB/sec throughput to over 1GB/sec, the p99 latency is under 30 milliseconds.

9 CONCLUSIONS AND FUTURE WORK

We present a system called Vortex for highly scalable ingestion and low latency retrieval of streaming data for real-time analysis. Vortex supports long running coarse and short running fine-grained transactions, allowing a variety of applications to be built over a unified API. Vortex was built based on the idea that rich insights are possible by combining streaming data and batch data into a single storage system. Furthermore, instead of adapting a batch system for streaming where we run into fundamental challenges around metadata overhead, we built the system for streaming first, and are able to tune it easily for the cost tradeoffs of batch scenarios. Vortex supports semi-structured data types, making it ideal for log analytics. Vortex brings streaming directly into the data warehouse. This allows applications to query their streaming and batch data through a expressive SQL interface. In BigQuery, we have been using Vortex for the last few years for both batch and streaming scenarios.

While Vortex provides competitive performance at the scale of ingestion that it supports, we see potential for further optimizations in data partitioning, resource consumption and tail latencies. We continue to push the envelope in these aspects for real time data. At the scales at which Vortex operates, in a multi-tenant environment, caching is a challenging problem. For some streaming applications, the most recent data is also the most interesting to read. Colossus already provides caching, but we are looking into further avenues to build query aware caching on top of our ingestion servers. We see Vortex as a building block for continuous stream analytics, as well as fine grained metadata management that we introduced in [8]. We also see opportunities in the monitoring space where low latency is preferred over 100% data availability.

ACKNOWLEDGMENTS

Vortex was the collective effort of many engineers and it builds on many services in Google's infrastructure stack. We would like to thank Jordan Tigani, Mosha Pasumansky, Andrew Fikes, Alex Lloyd, Sanjay Ghemawat, Tyler Akidau and Reuven Lax on their inputs into the early designs of Vortex, it's API and its interaction with other products in the ecosystem. We thank Sam McVeety and Chris Taylor for their review of the paper. We thank the following folks for their major contributions to Vortex's day to day development and operations: Yi Yang, Pavel Padinker, Teng Gu, Anastasia Han, Yaling Wu and Haochuan Fan. We also thank Victor Agababov for extending Vortex to support BigLake managed tables.

REFERENCES

[1] [n.d.]. The Elastic Stack. https://www.elastic.co/guide/index.html

[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 671–682. https://doi.org/10.1145/1142473.1142548

[3] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. 2016. Slicer: Auto-Sharding for Datacenter Applications.

[4] Hossein Ahmadi. 2016. *In-memory query execution in Google BigQuery*. https://cloud.google.com/blog/products/products/bigquery/in-memory-query-execution-in-google-bigquery

[5] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment* 8 (2015), 1792–1803.

[6] Eric Brewer. 2017. *Spanner, TrueTime and the CAP Theorem*. Technical Report.

[7] Purvi Desai and Kevin Leong. 2023. Rockset Concepts, Design and Architecture. https://rockset.com/Rockset_Concepts_Design_Architecture.pdf

[8] Pavan Edara and Mosha Pasumansky. 2021. Big Metadata: When Metadata is Big Data. *PVLDB* 14, 12 (2021), 3083–3095.

[9] Raúl Gracia-Tinedo, Flavio Junqueira, Tom Kaitchuck, and Sachin Joshi. 2023. Pravega: A Tiered Storage System for Data Streams. In *Proceedings of the 24th International Middleware Conference (Middleware '23)*. Association for Computing Machinery, New York, NY, USA, 165–177. https://doi.org/10.1145/3590140.3629113

[10] Dean Hildebrand and Denis Serenyi. 2021. *Colossus under the hood: a peek into Google's scalable storage system*. https://tinyurl.com/google-colossus

[11] Jay Kreps. 2011. Kafka : a Distributed Messaging System for Log Processing. https://api.semanticscholar.org/CorpusID:18534081

[12] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3461–3472. https://doi.org/10.14778/3415478.3415568

[13] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, and Google Inc. 2010. Dremel: Interactive Analysis of Web-Scale Datasets.

[14] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 476–487.

[15] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[16] Mosha Pasumansky. 2016. Inside Capacitor, BigQuery's Next-Generation Columnar Storage Format. Google Cloud Blog, Apr 2016. https://cloud.google.com/blog/products/bigquery/inside-capacitor-bigquerys-next-generation-columnar-storage-format

[17] Purujit Saha. 2020. *BigQuery under the hood: How zone assignments work*. https://cloud.google.com/blog/products/data-analytics/how-bigquery-zone-assignments-work/

[18] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France.

[19] Stephanie Wang. 2022. *BigQuery Write API explained*. https://tinyurl.com/bq-write-api-explained

[20] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*.

[21] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737