Marmaray: An Open Source Generic Data Ingestion and Dispersal Framework and Library for Apache Hadoop

September 12, 2018

By Danny Chen, Omkar Joshi

Connecting users worldwide on our platform all day, every day requires an enormous amount of data management. When you consider the hundreds of operations and data science teams analyzing large sets of anonymous, aggregated data, using a variety of different tools to better understand and maintain the health of our dynamic marketplace, this challenge is even more daunting.

Three years ago, Uber adopted the open source Apache Hadoop framework as its data platform, making it possible to manage petabytes of data across computer clusters. However, given our many teams, tools, and data sources, we needed a way to reliably ingest and disperse data at scale throughout our platform.

Enter Marmaray, Uber's open source, general-purpose Apache Hadoop data ingestion and dispersal framework and library. Built and designed by our Hadoop Platform team, Marmaray is a plug-in-based framework built on top of the Hadoop ecosystem. Users can add support to ingest data from any source and disperse to any sink leveraging the use of Apache Spark. The name, Marmaray, comes from a tunnel in Turkey connecting Europe and Asia. Similarly, we envisioned Marmaray within Uber as a pipeline connecting data from any source to any sink depending on customer preference.

Data lakes often hold data of widely varying quality. Marmaray ensures that all ingested raw data conforms to an appropriate source schema, maintaining a high level of quality so that analytical results are reliable. Data scientists can spend their time extracting useful insights from this data instead of handling data quality issues.

At Uber, Marmaray connects a collection of systems and services in a cohesive manner to do the following:

1. Produce quality schematized data through our schema management library and services.
2. Ingest data from multiple data stores into our Hadoop data lake via Marmaray ingestion.
3. Build pipelines using Uber's internal workflow orchestration service to crunch and process the ingested data as well as store and calculate business metrics based on this data in Hive.
4. Serve the processed results from Hive to an online data store where internal customers can query the data and get near-instantaneous results via Marmaray dispersal.

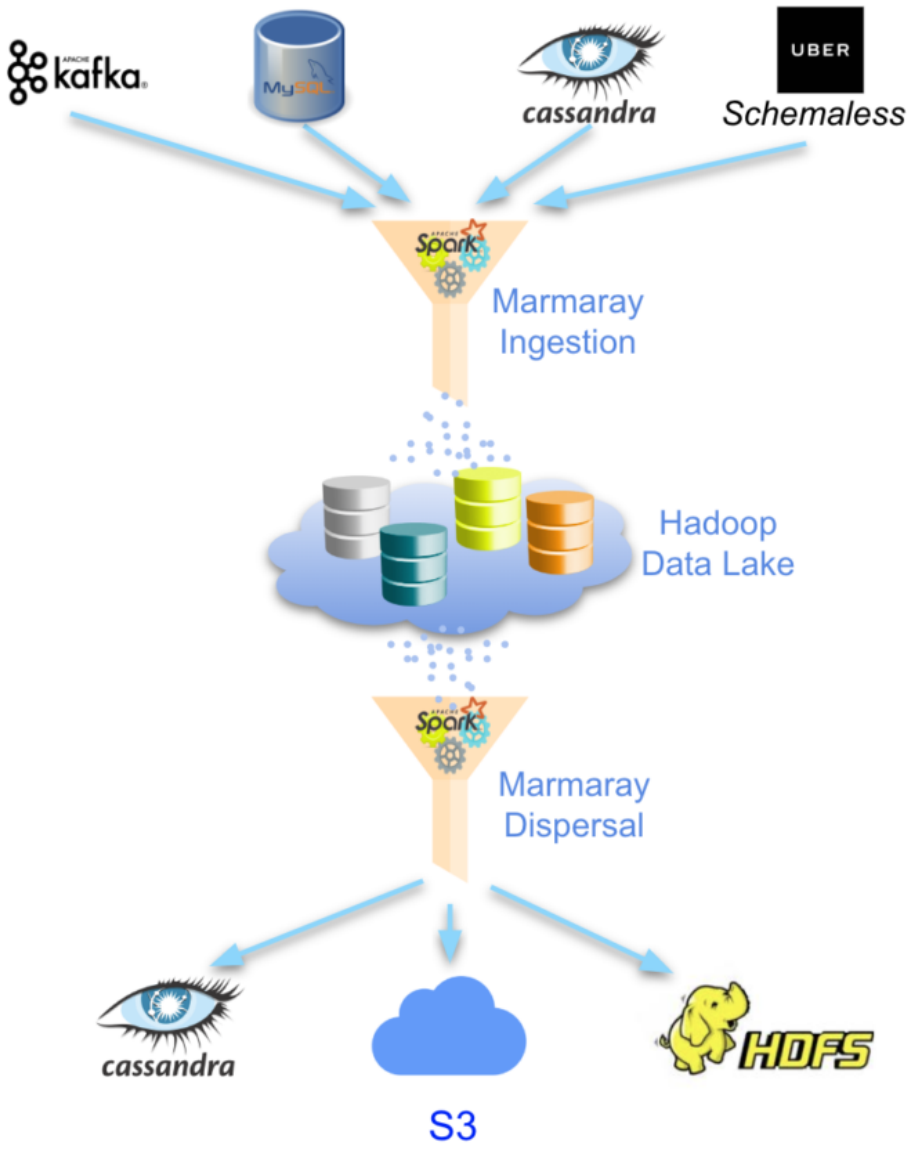

Figure 1: Marmaray both ingests data into our Hadoop Data Lake and disperses data to data stores. (Apache Kafka, Cassandra, Spark, and HDFS logos are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries. No endorsement by The Apache Software Foundation is implied by the use of these marks.)

While Marmaray realizes our vision of an any-source to any-sink data flow, we also found the need to build a completely self-service platform, providing users from a variety of backgrounds, teams, and technical expertise a frictionless onboarding experience.

The open source nature of Hadoop allowed us to integrate it into our platform for large-scale data analytics. As we built Marmary to facilitate data ingestion and dispersal on Hadoop, we felt it should also be turned over to the open source community. We hope that Marmaray will serve the data needs of other organizations, and that open source developers will broaden its functionality.

Ingestion challenges at scale

Uber's business generates a multitude of raw data, storing it in a variety of sources, such as Kafka, Schemaless, and MySQL. In turn, we need to ingest that data into our Hadoop data lake for our business analytics. The scale of data ingestion has grown exponentially in lock-step with the growth of Uber's many business verticals. The need for reliability at scale made it imperative that we re-architect our ingestion platform to ensure we could keep up with our pace of growth.

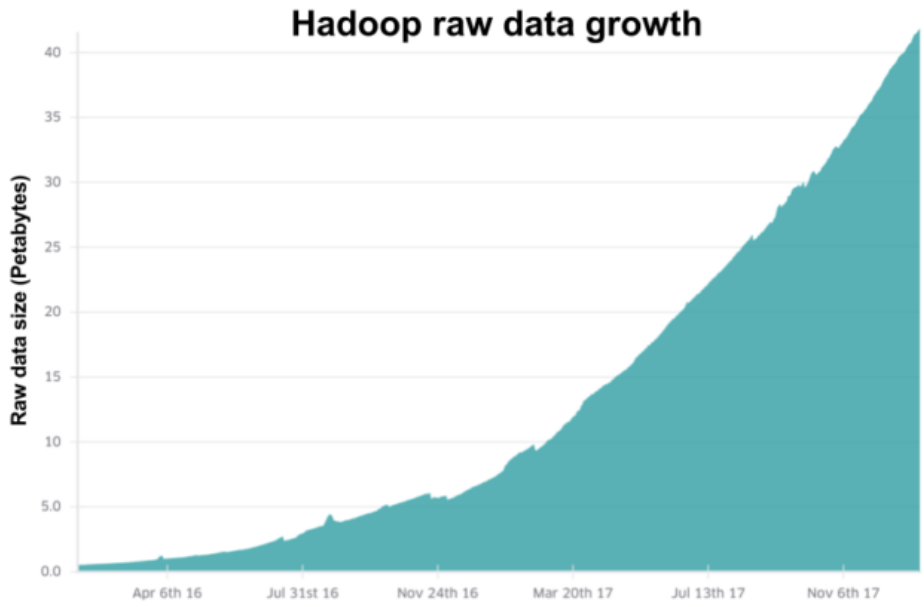

Figure 2: As Uber continued to expand our operations globally, raw data stored in our Hadoop data lake grew exponentially.

Our previous data architecture required running and maintaining multiple data pipelines, each corresponding to a different production codebase, which proved to be cumbersome over time as the amount of data increased. Data sources such as MySQL, Kafka, and Schemaless contained raw data that needed to be ingested into Hive to support diverse analytical needs from teams across the company. Each data source required understanding a different codebase and its associated intricacies as well as a different and unique set of configurations, graphs, and alerts. Adding new ingestion sources became non-trivial, and the overhead for maintenance required that our Big Data ecosystem support all of these systems. The on-call burden could be suffocating, with sometimes more than 200 alerts per week.

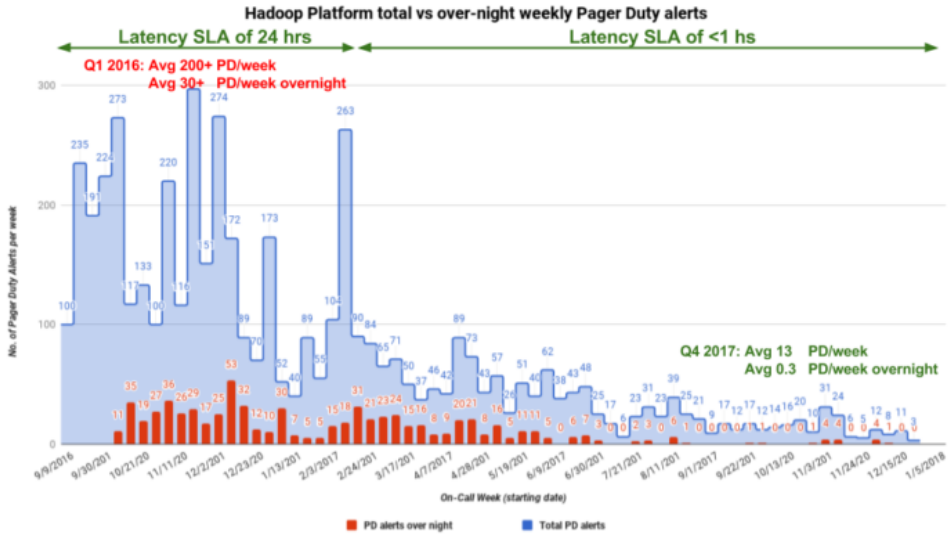

Figure 3: A graph of on-call alerts for our Hadoop Platform team illustrates the overhead involved with maintaining our systems.

With the introduction of Marmaray, we consolidated our ingestion pipelines into a single source-agnostic pipeline and codebase which will prove to be much more maintainable as well as resource-efficient.

This single ingestion pipeline will execute the same directed acyclic graph job (DAG) regardless of the source data store, where at runtime the ingestion behavior will vary depending on the specific source (akin to the strategy design pattern) to orchestrate the ingestion process and use a common flexible configuration suitable to handle future different needs and use cases.

Dispersal needs at Uber

Many of our internal data customers, such as the Uber Eats and Michelangelo Machine Learning platform teams, use Hadoop in concert with other tools to build and train their machine learning models to produce valuable, derived datasets to drive efficiencies and improve user experiences. To maximize the usefulness of these derived datasets, the need arose to disperse this data to online datastores, which often have much lower latency semantics than the Hadoop ecosystem.

Before we introduced Marmaray, each team was building their own ad-hoc dispersal systems. This duplication of efforts and creation of esoteric, non-universal features generally led to an inefficient use of engineering resources. Marmaray was envisioned, designed, and ultimately released in late 2017 to fulfill the need for a flexible, universal dispersal platform that would complete the Hadoop ecosystem by providing the means to transfer Hadoop data out to any online data store.

Tracking end-to-end data delivery

Many of our internal users need a guarantee that data generated from a source is delivered to a destination sink with a high degree of confidence. These same users also need completeness metrics covering how reliably the data has been delivered to the final sink. In theory this would mean 100 percent of the data has been delivered, but in practice we aim to deliver 99.99 to 99.999 percent of data to call the job complete. When the number of records are very small it is easy to run queries against source and sink systems to validate that data has been delivered.

At Uber, we ingest multiple petabytes of data and greater than 100 billion messages per day, so running these queries would not be possible. At this scale, we need a system which can track data delivery without a corresponding significant increase in latency. Marmaray uses a system to bucketize records via custom-authored accumulators in Spark, letting users monitor data delivery with minimum overhead.

Marmaray and Gobblin

Many of the fundamental building blocks and abstractions for Marmaray's design were inspired by Gobblin, a similar project developed at LinkedIn. The LinkedIn team was kind enough to share knowledge and provide a presentation about their project and architecture, which was greatly appreciated.

There are a couple of fundamental differences between Gobblin and Marmaray. While Gobblin is a universal data ingestion framework for Hadoop, Marmaray can both ingest data into and disperse data from Hadoop by leveraging Apache Spark. On the other hand, Gobblin leverages the Hadoop MapReduce framework to transform data, while Marmaray doesn't currently provide any transformation capabilities. Both frameworks provide easy-to-use self-service capabilities and can handle job and task scheduling and metadata management.

We chose Apache Spark as our main data processing engine instead of Hadoop MapReduce for a variety of reasons:

- Spark processes data much faster due to its in-memory processing semantics. This near real-time processing capability is critical for our service level agreements (SLAs).
- Spark's Resilient Distributed Datasets (RDDs) let us apply actions and other transformation such as filtering, mapping, and grouping to our data as well as data repartitioning and other optimizations to ensure our data flows maximize the processing capabilities of our Hadoop clusters.
- We can further leverage Spark to perform multiple data transformations without the need to store intermediate data to HDFS.
- We can take advantage of Spark's easy-to-use and familiar APIs for manipulating semi-structured data.
- Spark's native fault tolerance capabilities guarantee that random node failures and network partitions do not affect production capabilities.
- Given Spark's lower latency than MapReduce, it will let us eventually support Spark Streaming use cases to provide even tighter SLAs to our customers.
- Lazy evaluation of DAGs for data processing allows us to leverage more efficient optimization techniques, thereby reducing resource requirements, especially as we add DataFrame support in the future.
- Hadoop data is stored as Hudi format which is a storage abstraction library built on top of Spark.

Of course, with any design decision, trade-offs must be made. MapReduce and Spark are complementary frameworks, each better suited for specific needs. MapReduce can handle a much larger volume of data than Spark, which actually makes Gobblin a good fit for our scale. In practice and production, however, this has not been an issue. For Uber's SLA requirements, we found that Spark was a much better fit.

Marmaray use cases

Uber Eats

Let's take a look at how Marmaray helps improve the restaurant recommendations the Uber Eats app offers to its users. Based on an eater's order history, Uber Eats' machine learning models can recommend other restaurants that the eater might also enjoy.

The raw data from Uber Eats orders used by our prediction models is ingested into Hive through Marmaray. Machine learning models are then applied on top of this raw data to produce derived datasets of recommended restaurants, which are also persisted in Hive. If we accessed this data from Hive itself and surfaced these recommendations when the customer launches the app, the process would take on the order of seconds to possibly minutes resulting in a suboptimal customer

experience. By using Marmaray to disperse this data into a low-latency storage system like Cassandra, we are now able to access this information in milliseconds, resulting in a faster and more seamless user experience.

The ability to build technology that applies statistical techniques to extract intelligence from raw data is critical for Uber as we scale our growth. However, it is also critical that data exists in a store where it can be accessible in a timely manner to customers who need it. As the store of origin will often not meet the needs of customers, Marmaray's framework is crucial to ensuring this accessibility at scale.

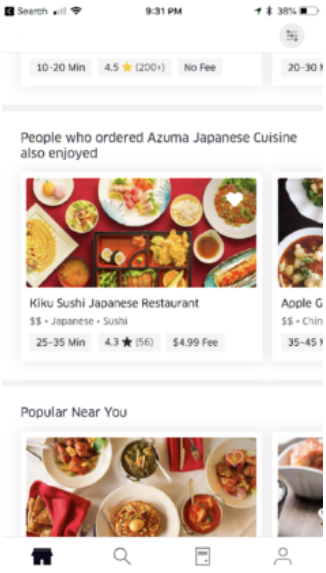

Figure 4: Through the app, Uber Eats uses machine learning models to recommend other restaurants an eater might enjoy.

Uber Freight

Uber Freight leverages Uber's platform to connect shippers with carriers and their drivers. Shippers can post loads they need moved, and drivers can pick loads they want to take. As a new business hosting most of its operational data in MySQL, Uber Freight lacked a means to run analytics.

Using Marmaray, Uber Freight was able to leverage a wealth of external data covering recency, relevance, lane information, and mileage, ingesting it into Freight Services. Ingesting millions of data points into Hadoop and distributing the data across tools used by multiple teams, Marmaray has helped Uber Freight's business scale quickly.

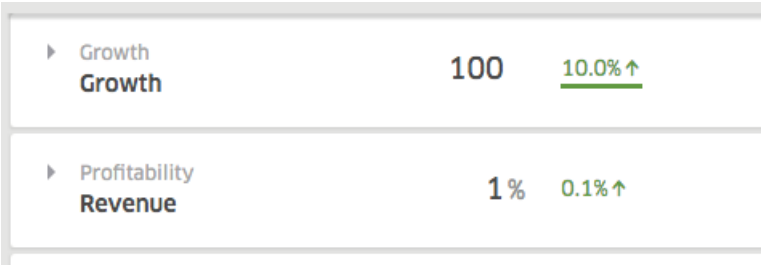

Figure 5: Using Marmaray, Uber Freight's metrics are updated every few hours, a much more responsive pace than the daily updates available previously. Sample data shown here does not reflect actual data, and is used for illustrative purposes only.

Marmaray architecture

The architecture diagram below illustrates the fundamental building blocks and abstractions in Marmaray that enable its overall job flow. These generic components facilitate the ability to add extensions to Marmaray, letting it support new sources and sinks.

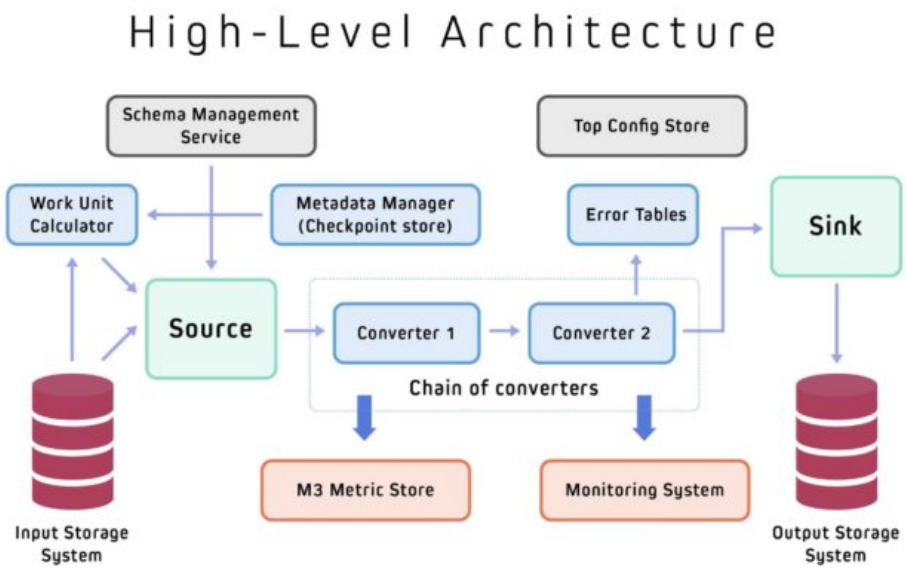

Figure 6: This diagram displays the high level architecture of the major components in the Marmaray framework.

DataConverters

Ingestion and dispersal jobs primarily perform transformations on input records from the source to ensure it is in the desired format before writing the data to the destination sink. Marmaray chains converters together to perform multiple transformations as needed, with the potential to also write to multiple sinks.

A secondary but critical function of DataConverters is to produce error records with every transformation. For analytical purposes, it is critical that all raw data conforms to a schema before it is ingested into our Hadoop data lake. Any data that is malformed, missing required fields, or otherwise deemed to have issues will be filtered out and written to error tables, ensuring a high level of data quality.

WorkUnitCalculator

Marmaray moves data in mini-batches of configurable size. In order to calculate the amount of data to process, we introduced the concept of a WorkUnitCalculator. At a very high level, the WorkUnitCalculator will look at the type of input source and the previously stored checkpoint, and then calculate the next work unit or batch of work. For example, a Work Unit could be Offset Ranges for Kafka or a collection of HDFS files for Hive/HDFS source.

When calculating the next batch of data to process, the WorkUnitCalculator can also take into account throttling information, for instance, the maximum amount of data to read or number of messages to read from Kafka. This throttling information is configurable per use case and offers maximum flexibility, thereby ensuring that work units are appropriately sized and don't overwhelm source or sink systems.

Metadata Manager

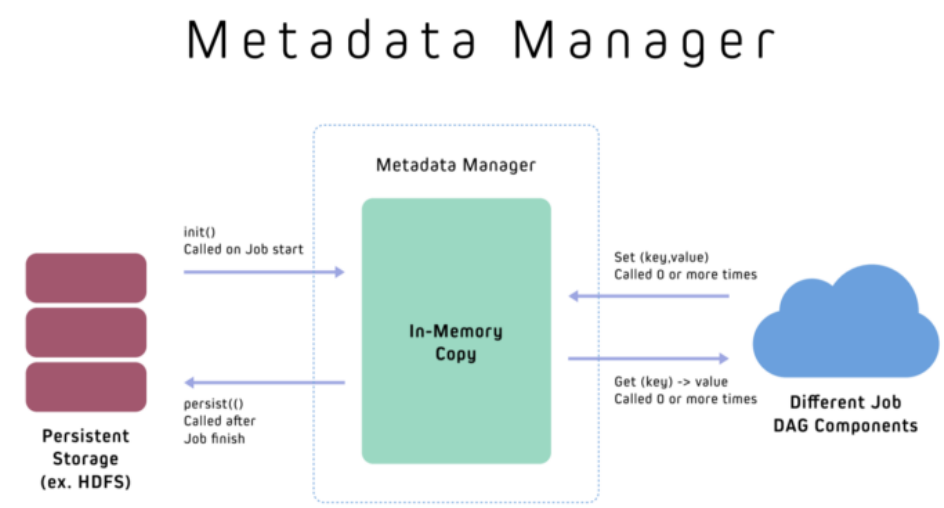

Figure 7: Marmaray's Metadata Manager is used to store any relevant metadata for a running job.

All Marmaray jobs need a persistent store, which we call the Metadata Manager, to cache job level metadata information. A job can update its state during its execution, and the job will replace the old saved state only if the current execution of the job is successful, otherwise, any modifications to the state are rejected. At Uber, we use the Metadata Manager to store such metadata as checkpoint information (or partition offsets in Kafka), average record size, and average number of messages. The metadata store is designed to be generic, however, and can store any relevant metrics that are useful to track, describe, or collect status on jobs depending on the use case and user needs.

ForkOperator and ForkFunction

Marmaray's ForkOperator abstraction splits the input stream of records using ForkFunction into multiple output streams. The canonical use case for ForkOperator is to have an input stream each for valid schema conforming records and error records, which then can be appropriately handled in a separate and independent manner.

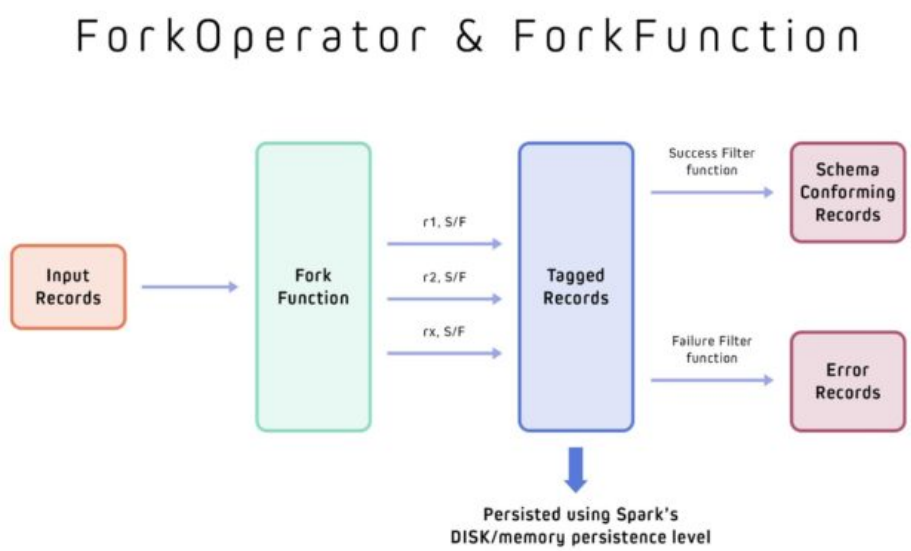

Figure 8: Fork Operator and Fork Function are used to split raw data records into a stream of schema conforming and error records to ensure high quality in our data lake.

ISource and ISink

The ISource contains all the necessary information from the source data for the appropriate requested work units, and ISink contains all the necessary information on how to write to the sink. For example, a Cassandra sink might contain information about the cluster, table, partitioning keys, and clustering keys for where the data should reside. A Kafka source would contain information about the topic name, maximum messages to read, cluster information, and offset initialization strategy, among other metadata.

Data model and job flow

The central component of Marmaray's architecture is what we call the AvroPayload, a wrapper around Avro's GenericRecord binary encoding format which includes relevant metadata for our data processing needs.

One of the major benefits of Avro data (GenericRecord) is that it is efficient both in its memory and network usage, as the binary encoded data can be sent over the wire with minimal schema overhead compared to JSON. Using Avro data running on top of Spark's architecture means we can also take advantage of Spark's data compression and encryption features. These benefits help our Spark jobs more efficiently handle data at a large scale.

Figure 9: AvroPayload wraps a GenericRecord with useful metadata.

To support our any-source to any-sink architecture, we require that all ingestion sources define converters from their schema format to Avro and that all dispersal sinks define converters from

the Avro Schema to the native sink data model (i.e., ByteBuffers for Cassandra).

Requiring that all converters either convert data to or from an AvroPayload format allows a loose and intentional coupling in our data model. Once a source and its associated transformation have been defined, the source theoretically can be dispersed to any supported sink, since all sinks are source-agnostic and only care that the data is in the intermediate AvroPayload format. We illustrate Marmaray's data model in Figure 8, below:

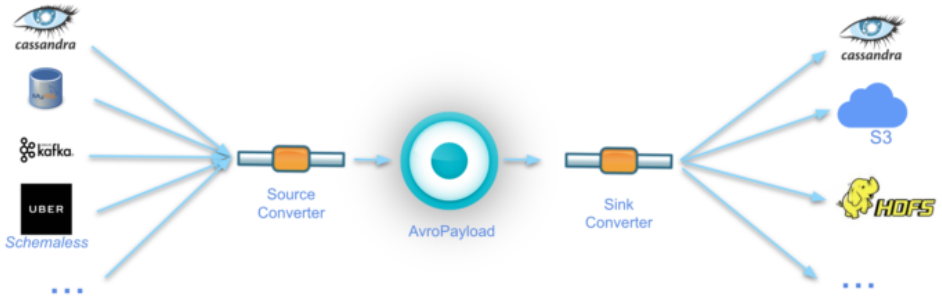

Figure 10: For ingestion and dispersal, Marmaray requires that data be converted into AvroPayload, a wrapper based on Avro's GenericRecord format. (Apache Kafka, Cassandra, Spark, and HDFS logos are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries. No endorsement by The Apache Software Foundation is implied by the use of these marks.)

Figure 11, below, gives a high level flow of how Marmaray jobs are orchestrated, independent of the specific source or sink.

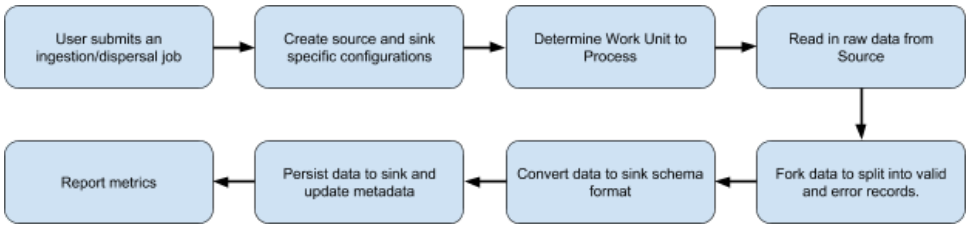

Figure 11: Marmaray runs its ingestion and dispersal jobs independent of source or sink, as illustrated by this job flow diagram.

During this process, the configuration defining specific attributes for each source and sink orchestrates every step of the next job. This includes figuring out the amount of data we need to process (i.e., its Work Unit), applying forking functions to split the raw data into 'valid' and 'error' records to ensure data quality, converting the data to an appropriate sink format, updating metadata, and reporting metrics to track progress. At Uber, all Marmaray jobs run on top of Apache Spark using YARN as the resource manager.

Self-serve platform

Since many of our data platform users are not familiar with the languages we use in our stack (for instance, Python and Java), it was important for our team to build out a self-service platform where users can simply point and click to set up an end-to-end pipeline, ensuring data from the desired source ends up in the correct sink to enable their analytical work and queries.

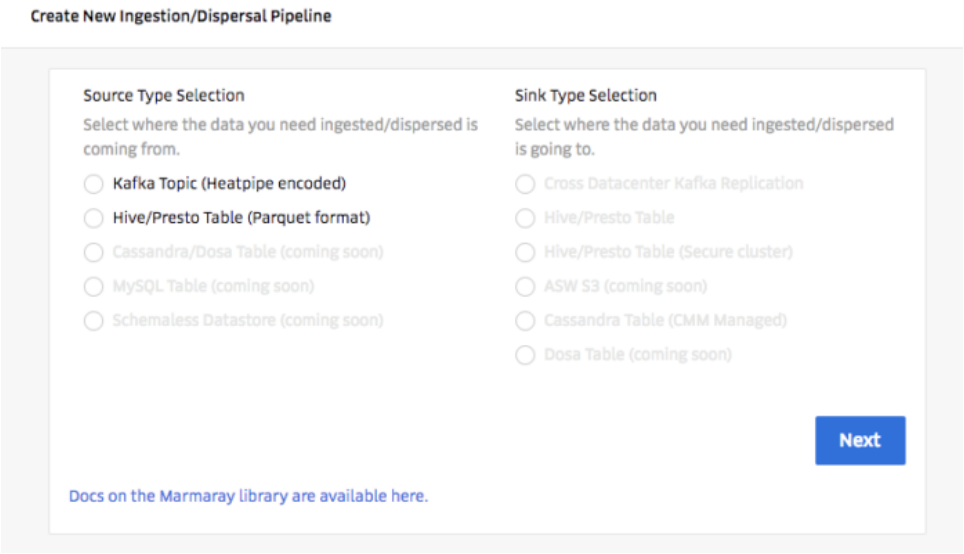

Figure 12: Our self-service UI enables data scientists and other users to move data from any source to any sink without having to know specific data formats.

In the seven months that our systems have been in production, over 3,300 jobs have been onboarded to our system through our self-service platform.

Data deletion

At Uber, all Kafka data is stored in append-only format with date-level partitions. The data for any specific user can span over multiple date partitions and will often have many Kafka records per partition. Scanning and updating all these partitions to correct, update, or delete user data can become very resource-intensive if the underlying storage doesn't include built-in indexing and update support. The Parquet data stores used by Hadoop don't support indexing, and we simply can't update Parquet files in place. To facilitate indexing and update support, Marmaray instead uses Hadoop Updates and Incremental (Hudi), an open source library also developed at Uber that manages storage of large analytical datasets to store the raw data in Hive.

At a high level, data producers scan the table using Hive, identify records to be deleted, and publish them to a Kafka cluster with user-specific information removed or masked. Marmaray's Kafka ingestion pipeline in turn reads them from the Kafka cluster, which has both new and updated (to-be-deleted) records. Marmaray then ingests pure new records using Hudi's bulk insert feature, keeping ingestion latencies low, and process updated records using Hudi's upsert feature to replace older Kafka records with newer modifications.

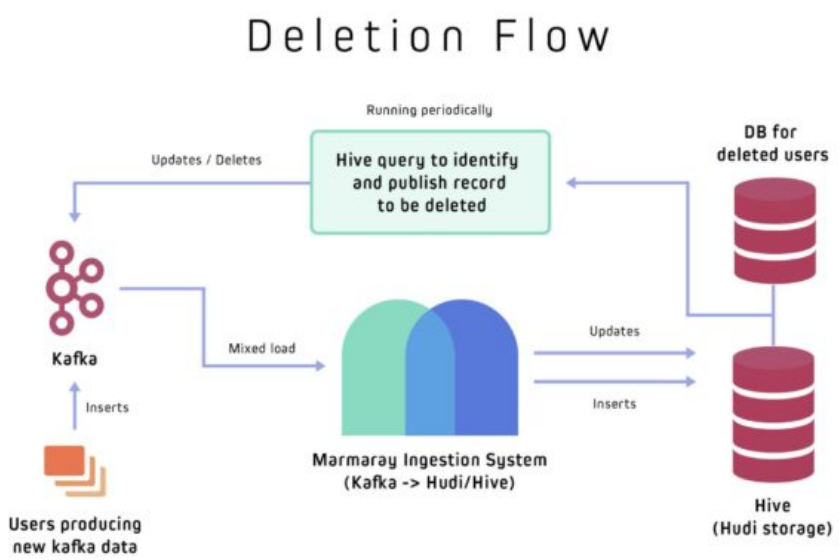

Figure 13: Marmaray also supports data deletion by leveraging the Hudi storage format.

Marmaray's next chapter

Marmaray's universal support of any-source to any-sink data pipelines is applicable to a wide range of use cases both in the Hadoop ecosystem (primarily when working with Hive) as well as for data migration. Along those lines, we have released Marmary to the open source community, and look forward to receiving feedback and contributions so that we can continually improve the Marmaray platform.

In the meantime, we are deprecating legacy pipelines and onboarding all our workflows onto Marmaray this year to simplify our overall data architecture and ensure that as our data needs increase we are able to scale as effortlessly as possible.

If you're interested in tackling data engineering challenges at scale, consider applying for a role on our team. For other Uber engineering opportunities, click here.

Subscribe to our newsletter to keep up with the latest innovations from Uber Engineering.

Blog Categories

AI
Architecture
Backend
Culture
Developers
General Engineering
Mobile
Open Source
Team Profile
Uber Data

Uber Links

Uber.com
Uber Eats
UberRUSH
Uber for Business
Help
Newsroom
Careers
Uber Open Source