# Uber's Big Data Platform: 100+ Petabytes with Minute Latency

October 17, 2018

*By Reza Shiftehfar*

1. unified
2. scale

— hadoop —

3. low latency

Uber is committed to delivering safer and more reliable transportation across our global markets. To accomplish this, Uber relies heavily on making data-driven decisions at every level, from forecasting rider demand during high traffic events to identifying and addressing bottlenecks in our driver-partner sign-up process. Over time, the need for more insights has resulted in over 100 petabytes of analytical data that needs to be cleaned, stored, and served with minimum latency through our Hadoop-based Big Data platform. Since 2014, we have worked to develop a Big Data solution that ensures data reliability, scalability, and ease-of-use, and are now focusing on increasing our platform's speed and efficiency.

In this article, we dive into Uber's Hadoop platform journey and discuss what we are building next to expand this rich and complex ecosystem.

## Generation 1: The beginning of Big Data at Uber

Before 2014, our limited amount of data could fit into a few traditional online transaction processing (OLTP) databases (in our case, MySQL and PostgreSQL). To leverage this data, our engineers had to access each database or table individually, and it was left to users to write their own code if they needed to combine data from different databases. At that time, we didn't have global access or a global view of all our stored data. In fact, our data was scattered across different OLTP databases, total data size was on the order of a few terabytes, and the latency to access this data was very fast (often, sub-minute). Figure 1, below, provides an overview of our data architecture prior to 2014:

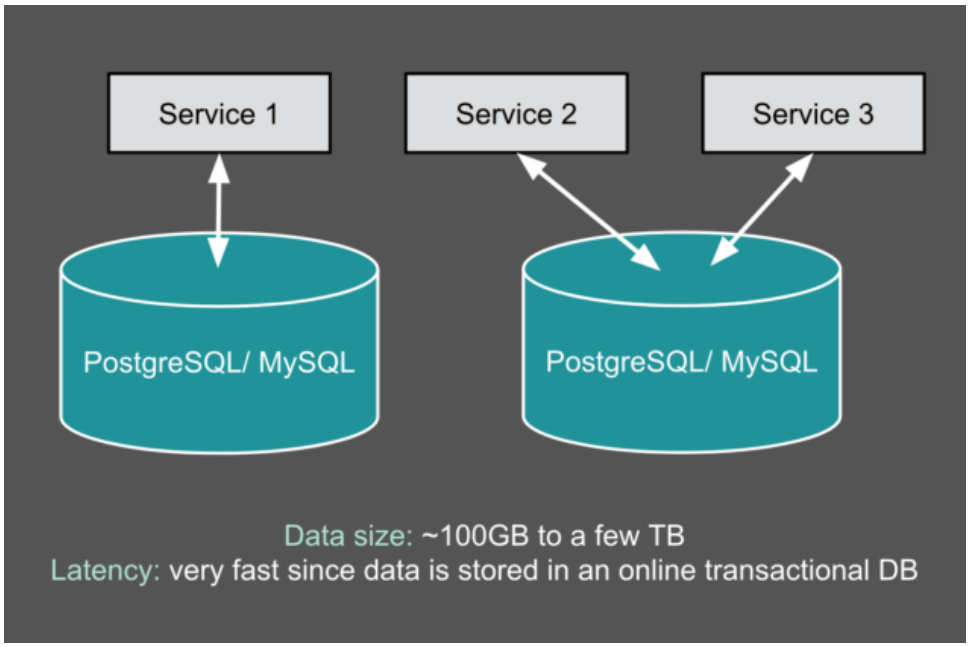


*Figure 1: Before 2014, the total amount of data stored at Uber was small enough to fit into a few traditional OLTP databases. There was no global view of the data, and data access was fast since each database was queried directly.*

With Uber's business growing exponentially (both in terms of the number of cities/countries we operated in and the number of riders/drivers using the service in each city), the amount of incoming data also increased and the need to access and analyze all the data in one place required us to build the first generation of our analytical data warehouse. To make Uber as data-driven as possible, we needed to ensure that analytical data was accessible to analysts, all in one place. To achieve this goal, we first categorized our data users into three main categories:

1. **City operations teams (thousands of users):** These on-the-ground crews manage and scale Uber's transportation network in each market. With our business expanding to new cities, there are thousands of city operations teams accessing this data on a regular basis to respond to driver-and-rider-specific issues.
2. **Data scientists and analysts (hundreds of users):** These are the analysts and scientists spread across different functional groups that need data to help deliver the best possible transportation and delivery experiences to our users, for instance, when forecasting rider demand to future-proof our services.
3. **Engineering teams (hundreds of users):** Engineers across the company focused on building automated data applications, such as our Fraud Detection and Driver Onboarding platforms.

The first generation of our analytical data warehouse focused on aggregating all of Uber's data in one place as well as streamlining data access. For the former, we decided to use Vertica as our data warehouse software because of its fast, scalable, and column-oriented design. We also developed multiple ad hoc ETL (Extract, Transform, and Load) jobs that copied data from different sources (i.e. AWS S3, OLTP databases, service logs, etc.) into Vertica. To achieve the latter, we standardized SQL as our solution's interface and built an online query service to accept user queries and submit them to the underlying query engine. Figure 2, below, depicts this analytical data warehouse:

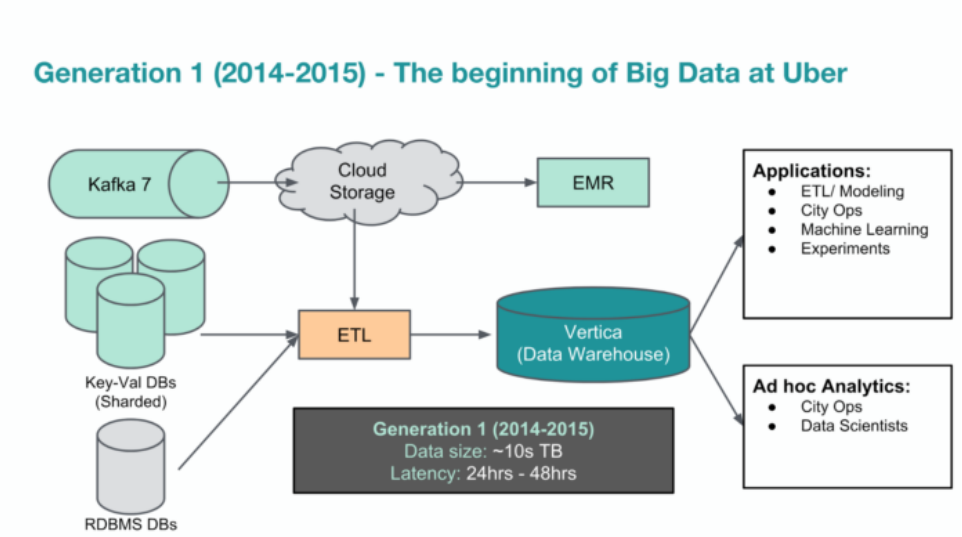


*Figure 2: The first generation of Uber's Big Data platform allowed us to aggregate all of Uber's data in one place and provide standard SQL interface for users to access data.*

Release of our first data warehousing service was a huge success for engineers across the company. For the first time, users had a global view and could access all data in one place. This resulted in a large number of new teams using data analysis as the foundation for their technology and product decisions. Within a few months, the size of our analytical data grew to tens of terabytes and the number of users increased to several hundred.

The use of SQL as a simple standard interface enabled city operators to easily interact with the data without knowing about the underlying technologies. In addition, different engineering teams started to build services and products tailored to user needs that were informed by this data (i.e. uberPool, upfront pricing, etc. ) and new teams were formed to better use and serve this data (i.e., our machine learning and experimentation teams).

## Limitations

On the other hand, the widespread use of our data warehouse and incoming data revealed a few limitations. Since data was ingested through ad hoc ETL jobs and we lacked a formal schema communication mechanism, data reliability became a concern. Most of our source data was in JSON format, and ingestion jobs were not resilient to changes in the producer code.

As our company grew, scaling our data warehouse became increasingly expensive. To cut down on costs, we started deleting older, obsolete data to free up space for new data. On top of this, much of our Big Data platform was not horizontally scalable since the primary goal was to unblock the critical business need for centralized data access or view and there was simply not enough time to ensure all parts were horizontally scalable. Our data warehouse was effectively being used as a data lake, piling up all raw data as well as performing all data modeling and serving the data.

Additionally, ETL jobs that ingested data into the data warehouse were also very fragile due to the lack of a formal contract between the services producing the data and the downstream data consumers (the use of flexible JSON format resulted in the lack of schema enforcement for the source data). The same data could be ingested multiple times if different users performed different transformations during ingestion. This resulted in extra pressure on our

upstream data sources (i.e., online data stores) and affected their quality of service. Moreover, this resulted in multiple copies of almost the same data being stored in our warehouse, further increasing storage costs. And in the case of data quality issues, backfilling was very time-and-labor-consuming because the ETL jobs were ad hoc and source-dependent, and data projections and transformation were performed during ingestion. It was also difficult to ingest any new data sets and types due to the lack of standardization in our ingestion jobs.

# Generation 2: The arrival of Hadoop

To address these limitations, we re-architected our Big Data platform around the Hadoop ecosystem. More specifically, we introduced a Hadoop data lake where all raw data was ingested from different online data stores only once and with no transformation during ingestion. This design shift significantly lowered the pressure on our online datastores and allowed us to transition from ad hoc ingestion jobs to a scalable ingestion platform. In order for users to access data in Hadoop, we introduced Presto to enable interactive ad hoc user queries, Apache Spark to facilitate programmatic access to raw data (in both SQL and non-SQL formats), and Apache Hive to serve as the workhorse for extremely large queries. These different query engines allowed users to use the tools that best addressed their needs, making our platform more flexible and accessible.

To keep the platform scalable, we ensured all data modeling and transformation only happened in Hadoop, enabling fast backfilling and recovery when issues arose. Only the most critical modeled tables (i.e., those leveraged by city operators in real time to run pure, quick SQL queries) were transferred to our data warehouse. This significantly lowered the operational cost of running a huge data warehouse while also directing users to Hadoop based query engines that were designed with their specific needs in mind.

We also leveraged the standard columnar file format of Apache Parquet, resulting in storage savings given the improved compression ratio and compute resource gains given the columnar access for serving analytical queries. Moreover, Parquet's seamless integration with Apache Spark made this solution a popular choice for accessing Hadoop data. Figure 3, below, summarizes the architecture of our second generation Big Data platform:

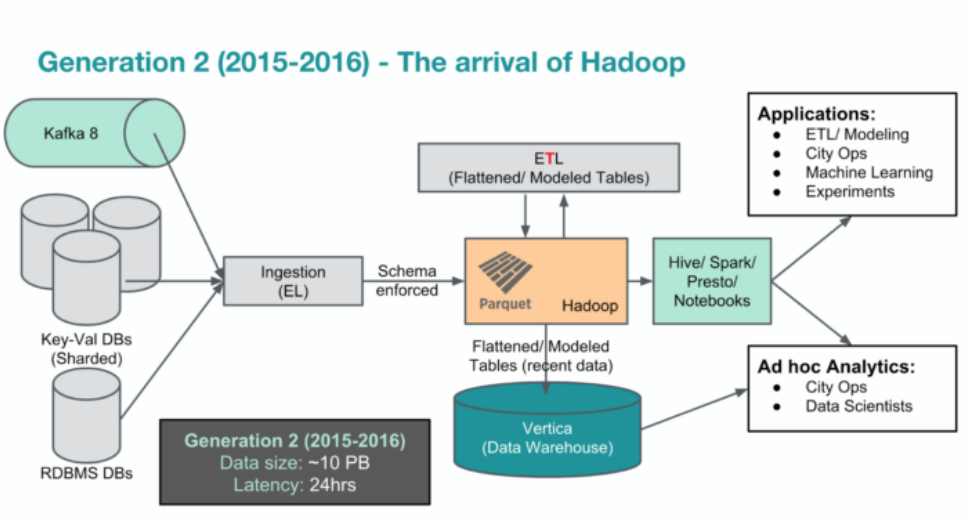


*Figure 3: The second generation of our Big Data platform leveraged Hadoop to enable horizontal scaling. Incorporating technologies such as Parquet, Spark, and Hive, tens of petabytes of data was ingested, stored, and served.*

In addition to incorporating a Hadoop data lake, we also made all data services in this ecosystem horizontally scalable, thereby improving the efficiency and stability of our Big Data platform. In particular, having this universal horizontal scalability to address immediate business needs allowed us to focus our energy on building the next generation of the data platform as opposed to ad hoc problem solving.

Unlike the first generation of our platform in which data pipelines were vulnerable to upstream data format changes, our second iteration allowed us to schematize all data, transitioning from JSON to Parquet to store schema and data together. To accomplish this, we built a central schema service to collect, store, and serve schemas as well as different client libraries to integrate different services with this central schema service. Fragile, ad hoc data ingestions jobs were replaced with a standard platform to transfer all source data in its original, nested format into the Hadoop data lake. Any required operations on and transformation of the data happened after ingestion via horizontally scalable batch jobs in Hadoop.

With Uber's business continuing to scale at light speed, we soon had tens of petabytes of data. On a daily basis, there were tens of terabytes of new data added to our data lake, and our Big Data platform grew to over 10,000 vcores with over 100,000 running batch jobs on any given day. This resulted in our Hadoop data lake becoming the centralized source-of-truth for all analytical Uber data.

## Limitations

As the company continued scaling and with tens of petabytes of data stored in our ecosystem, we faced a new set of challenges.

To start, the massive amount of small files stored in our HDFS (resulting from more data being ingested as well as more users writing ad hoc batch jobs which generated even more output data) began adding extra pressure on HDFS NameNodes. On top of that, data latency was still far from what our business needed. New data was only accessible to users once every 24 hours, which was too slow to make real-time decisions. While moving ETL and modeling into Hadoop made this process more scalable, these steps were still bottlenecks since these ETL jobs had to

recreate the entire modeled table in every run. Adding to the problem, both ingestion of the new data and modeling of the related derived table were based on creating new snapshots of the entire dataset and swapping the old and new tables to provide users with access to fresh data. The ingestion jobs had to return to the source datastore, create a new snapshot, and ingest or convert the entire dataset into consumable, columnar Parquet files during every run. With our data stores growing, these jobs could take over twenty hours with over 1,000 Spark executors to run.

A big part of each job involved converting both historical and new data from the latest snapshot. While only over 100 gigabytes of new data was added every day for each table, each run of the ingestion job had to convert the entire, over 100 terabyte dataset for that specific table. This was also true for ETL and modeling jobs that recreated new derived tables on every run. These jobs had to rely on snapshot-based ingestion of the source data because of the high ratio of updates on historical data. By nature, our data contains a lot of update operations (i.e., rider and driver ratings or support fare adjustments a few hours or even days after a completed trip). Since HDFS and Parquet do not support data updates, all ingestion jobs needed to create new snapshots from the updated source data, ingest the new snapshot into Hadoop, convert it into Parquet format, and then swap the output tables to view the new data. Figure 4, below, summarizes how these snapshot-based data ingestions moved through our Big Data platform:

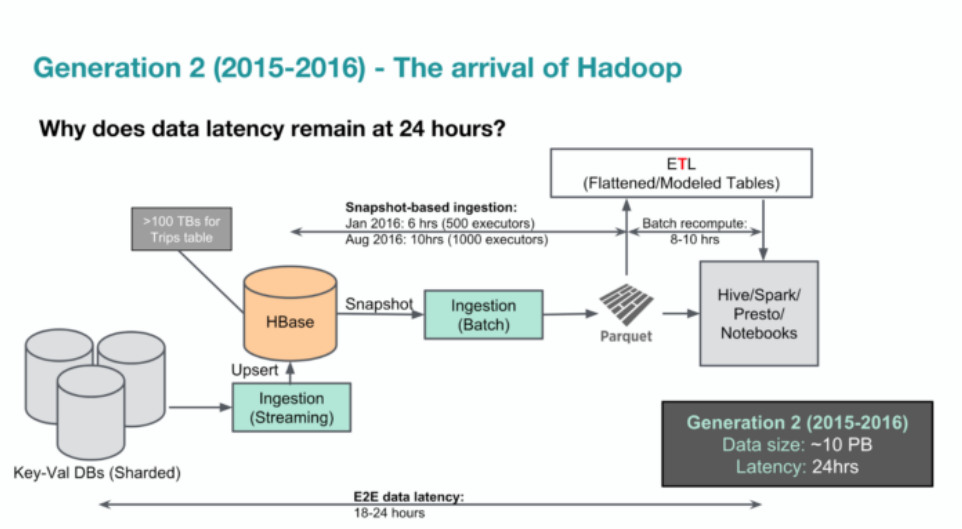


*Figure 4: While Hadoop enabled the storage of several petabytes of data in our Big Data platform, the latency for new data was still over one day, a lag due to the snapshot-based ingestion of large, upstream source tables that take several hours to process.*

# Generation 3: Rebuilding our Big Data platform for the long term

By early 2017, our Big Data platform was used by engineering and operations teams across the company, enabling them to access new and historical data all in one place. Users could easily access data in Hive, Presto, Spark, Vertica, Notebook, and more warehouse options all through a single UI portal tailored to their needs. With over 100 petabytes of data in HDFS, 100,000 vcores in our compute cluster, 100,000 Presto queries per day, 10,000 Spark jobs per day, and 20,000

Hive queries per day, our Hadoop analytics architecture was hitting scalability limitations and many services were affected by high data latency.

Fortunately, since our underlying infrastructure was horizontally scalable to address the immediate business needs, we had enough time to study our data content, data access patterns, and user-specific requirements to identify the most pressing concerns before building the next generation. Our research revealed four main pain points:

1. **HDFS scalability limitation:** This issue is faced by many companies who rely on HDFS to scale their big data infrastructures. By design, HDFS is bottlenecked by its NameNode capacity, so that storing large numbers of small files can significantly affect performance. This limitation usually occurs when data size grows beyond ten petabytes and becomes a real issue beyond 50-100 petabytes. Fortunately, there are relatively straightforward solutions to scale HDFS from a few tens to a few hundreds of petabytes, for instance leveraging ViewFS and using HDFS NameNode Federation. By controlling the number of small files and moving different parts of our data to separate clusters (e.g., HBase and Yarn app logs moved into a separate HDFS cluster), we were able to mitigate this HDFS limitation.
2. **Faster data in Hadoop:** Uber's business operates in real time and as such, our services require access to data that is as fresh as possible. As a result, 24-hour data latency was way too slow for many use cases and there was huge demand for faster data delivery. Our second generation Big Data platform's snapshot-based ingestion method was inefficient and prevented us from ingesting data with lower latency. To speed up data delivery, we had to re-architect our pipeline to the incremental ingestion of only updated and new data.
3. **Support of updates and deletes in Hadoop and Parquet:** Uber's data contains a lot of updates, ranging in age from the past few days (e.g., a rider or driver-partner adjusting a recent trip fare) to a few weeks (e.g., a rider rating their last trip the next time they take a new trip) or even a few months (e.g., backfilling or adjusting past data due to a business need). With snapshot-based ingestion of data, we ingest a fresh copy of the source data every 24 hours. In other words, we ingest all updates at one time, once per day. However, with the need for fresher data and incremental ingestion, our solution must be able to support update and delete operations for existing data. However, since our Big Data is stored in HDFS and Parquet, it is not possible to directly support update operations on the existing data. On the other hand, our data contains extremely wide tables (around 1,000 columns per table) with five or more levels of nesting while user queries usually only touch a few of these columns, preventing us from using non-columnar formats in a cost-efficient way. To prepare our Big Data platform for long-term growth, we had to find a way to solve this limitation within our HDFS file system so that we can support update/delete operations too.
4. **Faster ETL and modeling:** Similar to raw data ingestion, ETL and modeling jobs were snapshot-based, requiring our platform to rebuild derived tables in every run. To reduce data latency for modeled tables, ETL jobs also needed to become incremental. This required the ETL jobs to incrementally pull out only the changed data from the raw source table and update the previous derived output table instead of rebuilding the entire output table every few hours.

## Introducing Hudi

With the above requirements in mind, we built Hadoop Upserts anD Incremental (Hudi), an open source Spark library that provides an abstraction layer on top of HDFS and Parquet to

support the required update and delete operations. Hudi can be used from any Spark job, is horizontally scalable, and only relies on HDFS to operate. As a result, any Big Data platform that needs to support update/delete operations for the historical data can leverage Hudi.

Hudi enables us to update, insert, and delete existing Parquet data in Hadoop. Moreover, Hudi allows data users to incrementally pull out only changed data, significantly improving query efficiency and allowing for incremental updates of derived modeled tables.

Raw data in our Hadoop ecosystem is partitioned based on time and any of the old partitions can potentially receive updates at a later time. Thus, for a data user or an ETL job relying on these raw source data tables, the only way to know what date partition contains updated data is to scan the entire source table and filter out records based on some known notion of time. This results in a computationally expensive query requiring a full source table scan and prevents ETL jobs from running very frequently.

With Hudi, users can simply pass on their last checkpoint timestamp and retrieve all the records that have been updated since, regardless of whether these updates are new records added to recent date partitions or updates to older data (e.g., a new trip happening today versus an updated trip from 6 months ago), without running an expensive query that scans the entire source table.

Using the Hudi library, we were able to move away from the snapshot-based ingestion of raw data to an incremental ingestion model that enables us to reduce data latency from 24 hours to less than one hour. Figure 5, below, depicts our Big Data platform after the incorporation of Hudi:

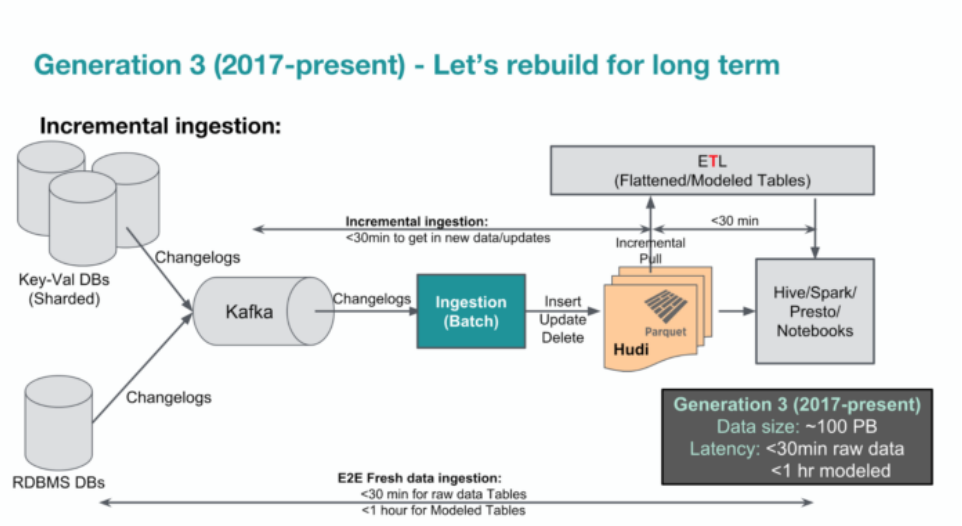


*Figure 5: The third generation of our Big Data platform incorporates faster, incremental data ingestion (using our open source Marmaray framework), as well as more efficient storage and serving of data via our open source Hudi library.*

# Generic data ingestion

Hudi isn't the only addition to the third generation of our Big Data platform. We also formalized the hand-over of upstream datastore changes between the storage and big data teams through Apache Kafka. Upstream datastore events (as well as classic logging messages from different

applications and services) stream into Kafka with a unified Avro encoding including standard global metadata headers attached (i.e., timestamp, row key, version, data center information, and originating host). Both the Streaming and Big Data teams use these storage changelog events as their source input data for further processing.

Our data ingestion platform, Marmaray, runs in mini-batches and picks up the upstream storage changelogs from Kafka, applying them on top of the existing data in Hadoop using Hudi library. As mentioned earlier, Hudi supports upsert operations, allowing users to add new records and update or delete historical data. Ingestion Spark jobs run every 10-15 minutes, providing a 30-minute raw data latency in Hadoop (having headroom for 1-2 ingestion job failures or retries). To avoid inefficiencies resulting from ingesting the same source data into Hadoop more than once, our setup does not allow any transformations during raw data ingestion, resulting in our decision to make our raw data ingestion framework an EL platform as opposed to a traditional ETL platform. Under this model, users are encouraged to perform desired transformation operations within Hadoop and in batch mode after upstream data lands in its raw nested format.

Since implementing these changes to our Big Data platform, we've saved a significant amount of computational resources by avoiding unnecessary or inefficient ingestion operations. The reliability of our raw data has also significantly improved, as we can now avoid error-prone transformations during ingestion. Now, users can run their transformations on top of raw source data using any Big Data processing engine. Moreover, in case of any issues, users can re-run their transformations again and still meet their SLAs by using more compute resources and a higher degree of parallelism to finish the batch transformation jobs faster.

## Incremental data modeling

Considering the large number of upstream data stores that need to be ingested into Hadoop (over 3,000 raw Hadoop tables as of 2017), we also built a generic ingestion platform that facilitates the ingestion of raw data into Hadoop in a unified and configurable way. Now, our Big Data platform updates raw Hadoop tables incrementally with a data latency of 10-15 minutes, allowing for fast access to source data. However, to ensure that modeled tables are also available with low latency, we must avoid inefficiencies (i.e., full derived table recreation or full source raw table scans) in our modeling ETL jobs too. In fact, Hudi allows ETL jobs to fetch only the changed data from the source table. Modeling jobs only need to pass a checkpoint timestamp during each iterative run to the Hudi reader to receive a stream of new or updated records from the raw source table (regardless of the date partition where the actual record is stored at).

The use of a Hudi writer during an ETL job enables us to update old partitions in the derived modeled tables without recreating the entire partition or table. Thus, our modeling ETL jobs use Hudi readers to incrementally fetch only the changed data from the source table and use Hudi writers to incrementally update the derived output table. Now, ETL jobs also finish in less than 30 minutes, providing end-to-end latency of less than one hour for all derived tables in Hadoop.

In order to provide data users of Hadoop tables with different options to access all data or only new or updated data, Hadoop raw tables using Hudi storage format provide two different reading modes:

1. **Latest mode view.** Provides a holistic view on the entire Hadoop table at that point in time. This view includes the latest merged values for all records as well as all the existing records in a table.

2. **Incremental mode view.** Fetches only the new and updated records from a specific Hadoop table based on a given timestamp. This view returns only the rows that have recently been inserted or have been updated since the latest checkpoint. Moreover, if a specific row is updated more than once since the last checkpoint, this mode returns all these intermediate changed values (rather than just returning the latest merged one)

Figure 6, below, depicts these two reading views for all Hadoop tables stored in Hudi file format:

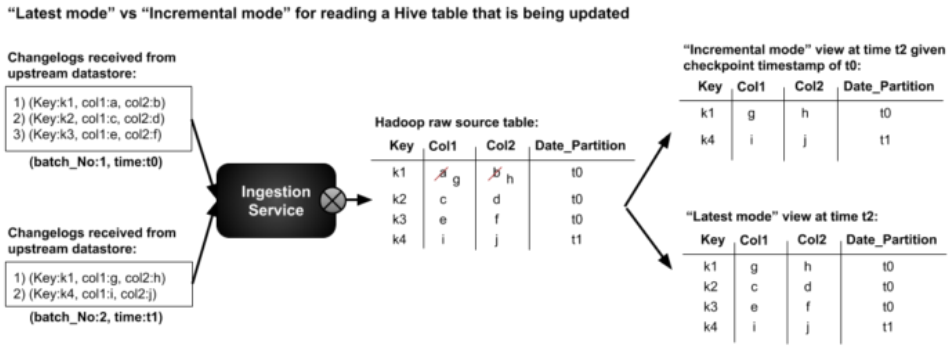


*Figure 6: A raw table that is being updated through Hudi writer can be read in two different modes: the latest mode view returning the latest value for all records and the incremental mode view returning only the updated records since last read.*

Users generally alternate between these two table views based on their needs. When they run an ad hoc query to analyze data based on the latest state, they use the latest mode view of the table (e.g., to fetch the total weekly number of trips per city in the U.S.). On the other hand, when a user has an iterative job or query that needs to fetch only changed or new records since its latest execution, they use the incremental mode view. Both views are available for all Hadoop tables at all times, and users can switch between different modes based on their needs.

## Standardized data model

In addition to providing different views of the same table, we also standardized our data model to provide two types of tables for all raw Hadoop data:

1. **Changelog history table**. Contains the history of all changelogs received for a specific upstream table. This table enables users to scan through the history of changes for a given table and can be merged per key to provide the latest value for each row.

2. **Merged snapshot table.** Houses the latest merged view of the upstream tables. This table contains the compacted merged view of all the historical changelogs received per key.

Figure 7, below, depicts how different Hive raw tables are generated for a specific upstream source datastore using the stream of given changelogs:

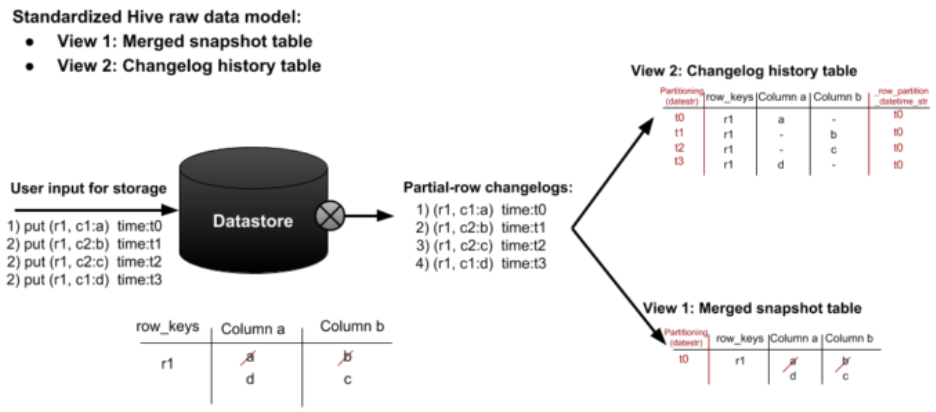


*Figure 7: Standardizing our Hive data model improved data quality for our entire Big Data ecosystem. This model incorporates a merged snapshot table containing the latest values for each row_key as well as a changelog history table containing the history of all value changes per each row_key.*

However, the stream of changelogs may or may not contain the entire row (all columns) for a given key. While merged snapshot tables always provide all the columns for a specific key, the changelog history table may be sparse if the upstream stream of changelogs only provides partial row changelogs, a functionality that improves efficiency by avoiding resending the entire row when only one or a few limited column values are changed. Should users want to fetch the changed values from the changelog history table and join it against the merged snapshot table to create the full row of data, we also include the date partition of the same key from the merged snapshot table in the changelog history table. This allows the two tables to more efficiently join across a specific partition by avoiding a full table scan of the merged snapshot table when joining the two.

Figure 8, below, summarizes the relationship between different components of our Big Data platform:

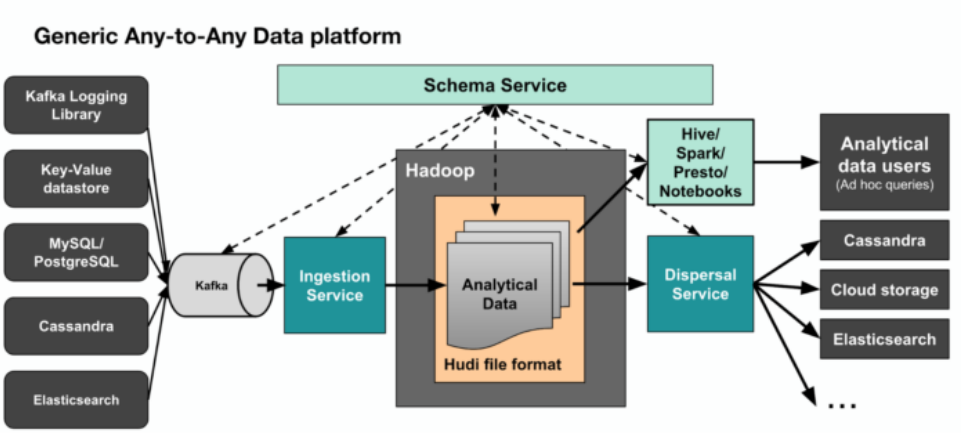


*Figure 8: Building a more extensible data transfer platform allowed us to easily aggregate all data pipelines in a standard way under one service as well as support any-to-any connectivity between any data source and data sink.*

# Generation 4: What's next?

Since rolling out the third generation of our Big Data platform in 2017, users across the company can quickly and reliably access data in Hadoop, but there is always room to grow. Below we summarize our ongoing efforts to enhance Uber's Big Data platform for improved data quality, data latency, efficiency, scalability, and reliability.

## Data quality

To enhance data quality, we identified two key areas for improvement. First, we want to avoid non-schema-conforming data when some of the upstream data stores do not mandatorily enforce or check data schema before storage (e.g., storing a key-value where the value is a JSON blob). This results in bad data entering our Hadoop ecosystem, thereby affecting all downstream users also relying on this data. To prevent an influx of bad data, we are transitioning all upstream data stores towards performing mandatory schema checks on data content and rejecting data entries if there are any issues (e.g., not confirming with the schema) with the data.

The second area that we found problematic was the quality of the actual data content. While using schemas ensures that data contains correct data types, they do not check the actual data values (e.g., an integer as opposed to a positive number between [0,150]). To improve data quality, we are expanding our schema service to support semantic checks. These semantic checks (in other words, Uber-specific data types) allows us to add extra constraints on the actual data content beyond basic structural type checking.

## Data latency

We are aiming to reduce raw data latency in Hadoop to five minutes and data latency for modeled tables to ten minutes. This will allow more use cases to move away from stream processing to more efficient mini-batch processing that uses Hudi's incremental data pulls.

We are also expanding our Hudi project to support an additional view mode, which will include the existing read-optimized view, as well as a new real-time view which shows data with latency of just a few minutes. This real-time view relies on an open source solution (and part of Hudi) we call Merge-On-Read or Hudi 2.0.

## Data efficiency

To improve data efficiency, we are moving away from relying on dedicated hardware for any of our services and towards service dockerization. In addition, we are unifying all of our resource schedulers within and across our Hadoop ecosystem to bridge the gap between our Hadoop and non-data services across the company. This allows all jobs and services to be scheduled in a unified fashion regardless of the medium it will be executed in. As Uber grows, data locality will be a big concern for Hadoop applications, and a successful unified resource manager can bring together all existing schedulers (i.e., Yarn, Mesos, and Myriad).

## Scalability and reliability

As part of our effort to improve the scalability and reliability of our platform, we identified several issues related to possible edge cases. While our ingestion platform was developed as a

generic, pluggable model, the actual ingestion of upstream data still includes a lot of source-dependent pipeline configurations, making the ingestion pipeline fragile and increasing the maintenance costs of operating several thousands of these pipelines.

To ensure we have unified data ingestion regardless of data source, we have started a project in collaboration with Uber's Data Storage team to unify the content, format, and metadata of the changelogs from all upstream data sources regardless of their technological makeup. This project will ensure that information about these specific upstream technologies will only be an additional metadata added to the actual changelog value (as opposed to having totally different changelog content and metadata for different data sources) and data ingestion will happen regardless of the upstream source.

Finally, our next version of Hudi will allow us to generate much larger Parquet files (over one gigabyte compared to our current 128 megabytes) by default within a few minutes for all of our data sources. It will also remove any sensitivities around the ratio of updates versus inserts. Hudi 1.0 relies on a technique called copy-on-write that rewrites the entire source Parquet file whenever there is an updated record. This significantly increases the write amplification, especially when the ratio of update to insert increases, and prevents creation of larger Parquet files in HDFs. The new version of Hudi is designed to overcome this limitation by storing the updated record in a separate delta file and asynchronously merging it with the base Parquet file based on a given policy (e.g., when there is enough amount of updated data to amortize the cost of rewriting a large base Parquet file). Having Hadoop data stored in larger Parquet files as well as a more reliable source-independent data ingestion platform will allow our analytical data platform to continue grow in the upcoming years as the business thrive.

# Moving forward

Uber's data organization is a cross-functional collaboration between the Data Platform, Data Foundation, Streaming and Real-time Platform, and Big Data teams to build the required libraries and distributed services that support Uber's analytical data infrastructure. If working on Big Data challenges that boggle the limits of scale interests you, consider applying for a role on our San Francisco and Palo Alto-based teams.

*Please email your résumé to hadoop-platform-jobs@uber.com if you are interested in working with us!*

*Reza Shiftehfar currently leads Uber's Hadoop Platform team. His team helps build and grow Uber's reliable and scalable Big Data platform that serves petabytes of data utilizing technologies such as Apache Hadoop, Apache Hive, Apache Kafka, Apache Spark, and Presto. Reza is one of the founding engineers of Uber's data team and helped scale Uber's data platform from a few terabytes to over 100 petabytes while reducing data latency from 24+ hours to minutes.*

Engineering Blog

Get the App →

Become a Driver →

Contact Us

ubereng@uber.com

@ubereng

Follow us

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



Privacy Policy

Terms and Conditions