Hudi: Uber Engineering's Incremental Processing Framework on Apache Hadoop

March 12, 2017

By Prasanna Rajaperumal & Vinoth Chandar

With the evolution of storage formats like Apache Parquet and Apache ORC and query engines like Presto and Apache Impala, the Hadoop ecosystem has the potential to become a general-purpose, unified serving layer for workloads that can tolerate latencies of a few minutes. In order to achieve this, however, it requires efficient and low latency data ingestion and data preparation in the Hadoop Distributed File System (HDFS).

To address this at Uber, we built Hudi (pronounced as "hoodie"), an incremental processing framework to power all business critical data pipelines at low latency and high efficiency. In fact, we've recently open sourced it for others to use and build on. But before diving into Hudi, let's take a step back and discuss why it's a good idea to think about Hadoop as the unified serving layer.

Motivation

Lambda architecture is a common data processing architecture that proposes double compute with streaming and batch layer. Once every few hours, a batch process is kicked off to compute the accurate business state and the batch update is bulk loaded into the serving layer. Meanwhile, a stream processing layer computes and serves the same state to circumvent the above multi-hour latency. However, this state is only an approximate one until it is overridden by the more accurate batch computed state. Since the states are slightly different, there needs to be either different serving layers for batch and stream, coalesced in an abstraction on top, or a rather complex serving system (like Druid) which performs reasonably well for record-level updates and batch bulk loads.

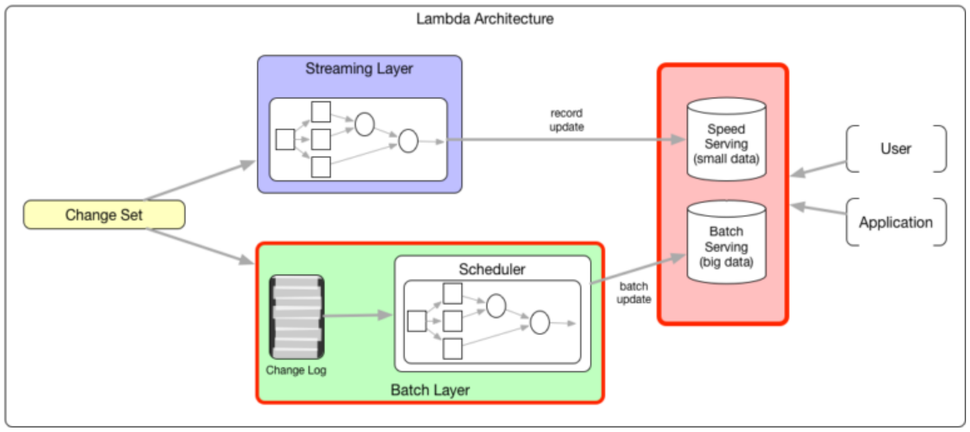

Figure 1: Lambda architecture requires double compute and double serving.

Questioning the need for a separate batch layer, Kappa architecture argues that a stream processing engine could be a general-purpose solution for computations. In a generic sense, all computations can be described as operators producing a tuple stream and consumers iterating over multiple input tuple streams (i.e. Volcano Iterator model). This functionality would enable the streaming layer to handle the reprocessing of business states by replaying computation with increased parallelism and resources. With systems that can efficiently checkpoint and store large amounts of streaming state, the business state in the streaming layer is no longer an approximation; this model has gained some traction with many ingest pipelines. Still, even though the batch layer is eliminated in this model, the problem of having two different serving layers remains.

Many true stream processing systems today operate at record level, so speed serving systems should be optimized for record-level updates. Typically, these systems cannot be optimized for analytical scans as well, unless the system has either a large chunk of its data in-memory (like Memsql) or aggressive indexes (like ElasticSearch). These speed-serving systems sacrifice scalability and cost for optimized ingest and scan performance. For this reason, data retention in these serving systems is typically limited, meaning that they can last 30 to 90 days or store up to a few TBs of data. Analytics on older historical data is often redirected to query engines on HDFS where data latency is not an issue.

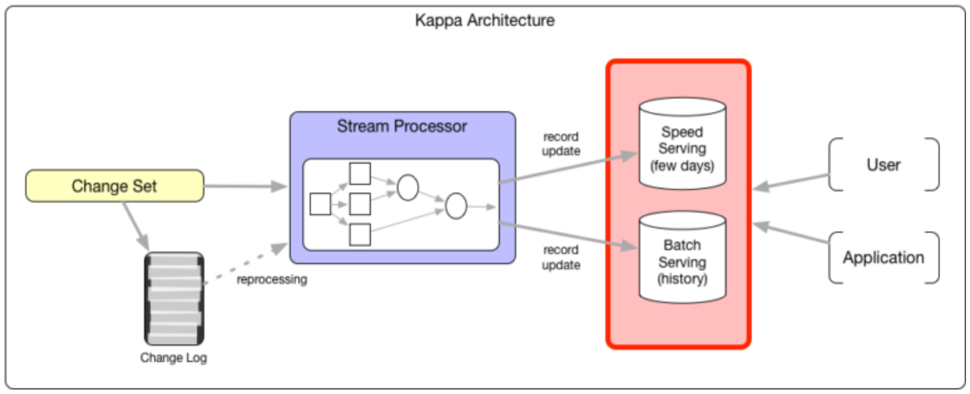

Figure 2: Kappa architecture simplifies computing by unifying processing, but serving complexity still exists.

This fundamental tradeoff between data ingest latency, scan performance, and compute resources and operational complexity is unavoidable. But for workloads that can tolerate latencies of about 10 minutes, there is no need for a separate "speed" serving layer if there is a faster way to ingest and prepare data in HDFS. This unifies the serving layer and reduces the overall complexity and resource usage significantly.

However, for HDFS to become the unified serving layer, it needs to not only store a log of changesets (a system of record), but also support compacted, de-duplicated business states partitioned by a meaningful business metric. The following features are required for this type of unified serving layer:

- Ability to quickly **apply mutations** to large HDFS datasets
- Data storage options that are **optimized for analytical scans** (columnar file formats)
- Ability to **chain and propagate updates** efficiently to modeled datasets

Compacted business state usually can't avoid mutations, even if the business partition field is the time in which the event occurred. Ingestion can still result in updates to many older partitions because of late-arriving data and the difference between event and processing times. Even if the partition key is the processing time, there may still be a need for updates because of the demand to wipe out data for audit compliance or security reasons.

Introducing Hudi: Hi, Hudi!

Enter Hudi, an incremental framework that supports the above requirements outlined in our previous section. In short, Hudi (**H**adoop **U**psert **D**elete and **I**ncremental) is an analytical, scan-optimized data storage abstraction which enables applying mutations to data in HDFS on the order of few minutes and chaining of incremental processing.

Hudi datasets integrate with the current Hadoop ecosystem (including Apache Hive, Apache Parquet, Presto, and Apache Spark) through a custom InputFormat, making the framework seamless for the end user.

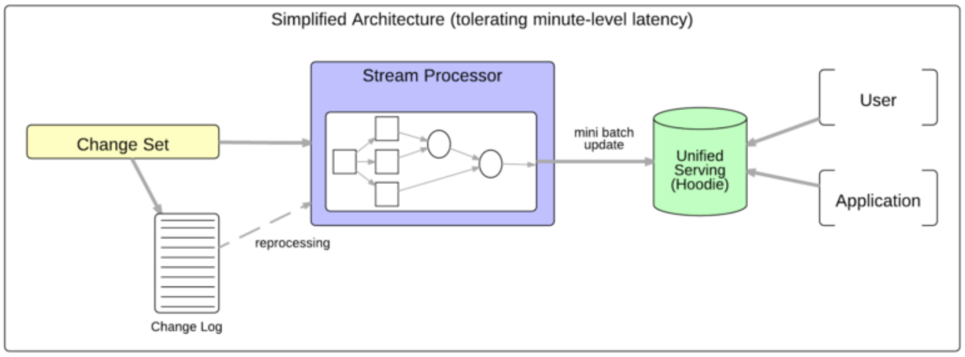

Figure 3: Hudi simplifies serving for workloads tolerating minute-level latency.

The DataFlow model characterizes data pipelines based on their latency and completeness guarantees. Figure 4, below, demonstrates how pipelines at Uber Engineering are distributed across this spectrum and what styles of processing are typically applied for each:

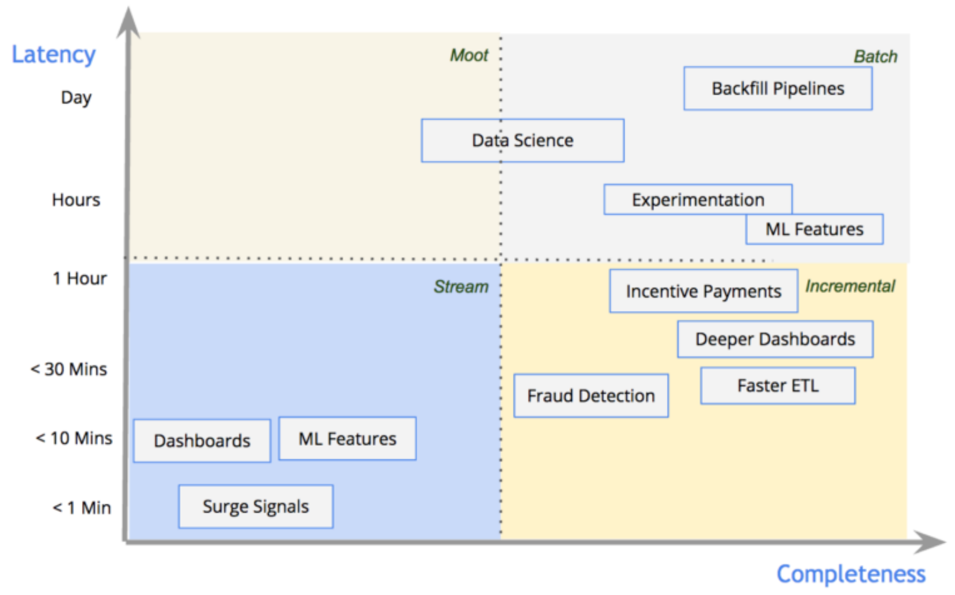

Figure 4: The above diagram demonstrates the distribution of use-cases across different latencies and completeness levels at Uber.

For the few use cases truly needing ~1 minute latencies and dashboards with simple business metrics, we rely on record-level stream processing. For traditional batch use cases like machine learning and experiment effectiveness analysis, we rely on batch processing that excels at heavier computations. For use cases where complex joins or significant data crunching is needed at near real-time latencies, we rely on Hudi and its incremental processing primitives to obtain the best of both worlds. To learn more about the use cases supported by Hudi, you can check out our documentation on Github.

Storage

Hudi organizes a dataset into a partitioned directory structure under a *basepath*, similar to a traditional Hive table. The dataset is broken up into partitions, which are directories containing data files for that partition. Each partition is uniquely identified by its *partitionpath* relative to the *basepath*. Within each partition, records are distributed into multiple data files. Each data file is identified by both an unique *fileId* and the commit that produced the file. In the case of updates, multiple data files can share the same *fileId* written at different commits.

Each record is uniquely identified by a record key and mapped to a *fileId*. This mapping between record key and *fileId* is permanent once the first version of a record has been written to a file. In short, the *fileId* identifies a group of files that contain all versions of a group of records.

Hudi storage consists of three distinct parts:

1. **Metadata**: Hudi maintains the metadata of all activity performed on the dataset as a timeline, which enables instantaneous views of the dataset. This is stored under a metadata directory in the *basepath*. Below we've outlined the types of actions in the timeline:
   - Commits: A single commit captures information about an atomic write of a batch of records into a dataset. Commits are identified by a monotonically increasing timestamp, denoting the start of the write operation.
   - Cleans: Background activity that gets rid of older versions of files in the dataset that will no longer be used in a running query.
   - Compactions: Background activity to reconcile differential data structures within Hudi (e.g. moving updates from row-based log files to columnar formats).

2. **Index**: Hudi maintains an index to quickly map an incoming record key to a *fileId* if the record key is already present. Index implementation is pluggable and the following are the options currently available:
   - Bloom filter stored in each data file footer: The preferred default option, since there is no dependency on any external system. Data and index are always consistent with one another.
   - Apache HBase: Efficient lookup for a small batch of keys. This option is likely to shave off a few seconds during index tagging.

3. **Data**: Hudi stores all ingested data in two different storage formats. The actual formats used are pluggable, but fundamentally require the following characteristics:
   - Scan-optimized columnar storage format (*ROFormat*). Default is Apache Parquet.
   - Write-optimized row-based storage format (*WOFormat*). Default is Apache Avro.

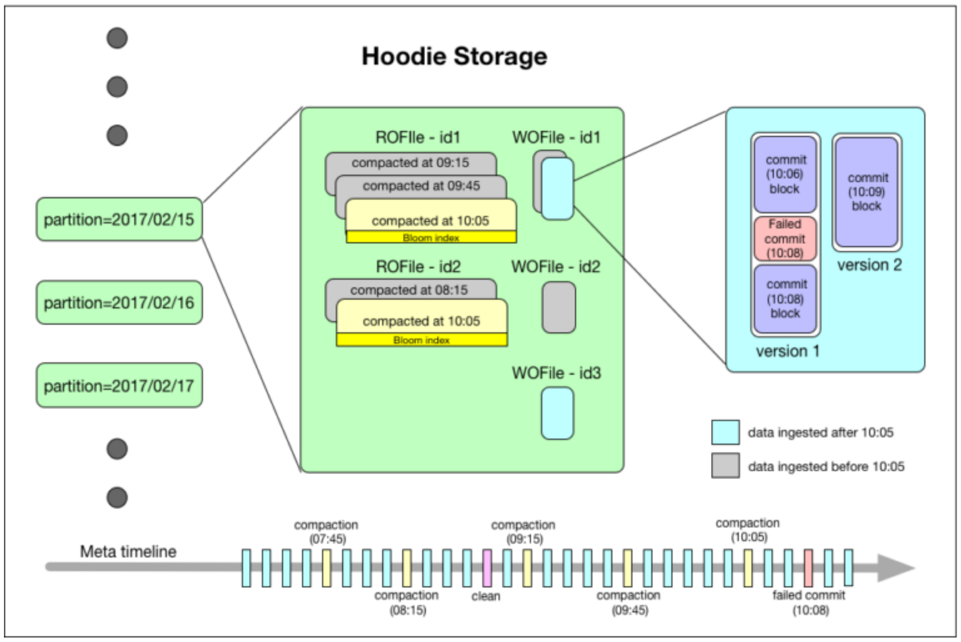

Figure 5: Hudi Storage Internals. The above Hudi Storage diagram depicts a commit time in YYYYMMDDHHMISS format and can be simplified as HH:SS.

Optimization

Hudi storage is optimized for HDFS usage patterns. Compaction is the critical operation to convert data from a write-optimized format to a scan-optimized format. Since the fundamental unit of parallelism for a compaction is rewriting a single *fileId*, Hudi ensures all data files are written out as HDFS block-sized files to balance compaction parallelism, query scan parallelism, and the total number of files in HDFS. Compaction is also pluggable, which can be extended to stitch older, less frequently updated data files to further reduce the total number of files.

Ingest Path

Hudi is a Spark library that is intended to be run as a streaming ingest job, and ingests data as mini-batches (typically on the order of one to two minutes). However, depending on latency requirements and resource negotiation time, the ingest jobs can also be run as scheduled tasks using Apache Oozie or Apache Airflow.

The following is the write path for a Hudi ingestion with the default configuration:

1. Hudi loads the Bloom filter index from all parquet files in the involved partitions (meaning, partitions spread from the input batch) and tags the record as either an update or insert by mapping the incoming keys to existing files for updates. The join here could skew on input batch size, partition spread, or number of files in a partition. It is handled automatically by doing a range partitioning on a joined key and sub-partitioned to avoid the notorious 2GB limit for a remote shuffle block in Spark.

2. Hudi groups inserts per partition, assigns a new *fileId*, and appends to the corresponding log file until the log file reaches the HDFS block size. Once the block size is reached, Hudi creates another *fileId* and repeats this process for all inserts in that partition.
    1. A time-limited compaction process is kicked off by a scheduler every few minutes, which generates a prioritized list of compactions and compacts all the avro files for a *fileId* with the current parquet file to create the next version of that parquet file.
    2. Compaction runs asynchronously, locking down specific log versions being compacted and writing new updates to that *fileId* into a new log version. Locks are obtained in Zookeeper.
    3. Compactions are prioritized based on the size of the log data being compacted, and are pluggable with a compaction strategy. On every compaction iteration, the files with the largest amount of logs are compacted first, while small log files are compacted last, since the cost of re-writing the parquet file is not amortized on the number of updates to the file.

3. Hudi appends updates for a *fileId* to its corresponding log file if one exists or creates a log file if one doesn't exist.

4. If the ingest job succeeds, a commit is recorded in the Hudi meta timeline, which atomically renames an inflight file to a commit file and writes out details about partitions and the *fileId* version created.

Optimization

As discussed before, Hudi strives to align file size with the underlying block size. Depending on the efficiency of columnar compression and the volume of data in a partition to compact, compaction can still create small parquet files. This is eventually autocorrected in the next iterations of ingestion, since inserts to a partition are packed as updates to existing small files. Eventually, file sizes will grow to reach the underlying block size on compaction.

Failure Recovery

If an ingest job fails because of an intermittent error, Spark retries computing the RDD and auto-resolves. If the number of failures exceed *maxRetries* in Spark, then the ingest job fails, and the next iteration will retry ingesting the same batch again. Two important distinctions are noted below:

- Failed ingestion can write partial avro blocks in the log file.
    - This is handled by storing metadata about the start offset for a block and log file version in the commit metadata. When reading the log, the irrelevant, at-times partially-written commit blocks are skipped and the seek location is set appropriately on the avro file.

- Failed compaction could write partial parquet files.
    - This is handled by the query layer, which filters file versions based on the commit metadata. The query layer will only pick files for the last completed compaction. The next compaction iteration will rollback the failed compaction and try again.

Query Path

The commit meta timeline enables both a read-optimized view and a realtime view of the same data in HDFS; these views let the client choose between data latency and query execution time. Hudi provides these views with a custom *InputFormat*, and includes a Hive registration module which registers both these views as Hive metastore tables. Both of these input formats understand *fileId* and commit time, and filters the files to pick only the most recently committed files. Then, Hudi generates splits on those data files to run the query plan. *InputFormat* details are outlined below:

- *HoodieReadOptimizedInputFormat*: Provides a scan-optimized view which filters out all the log files and just picks the latest versions of compacted parquet files.
- *HoodieRealtimeInputFormat*: Provides a more real-time view which, in addition to picking the latest versions of compacted parquet files, also provides a *RecordReader* to merge the log files with their corresponding parquet files during a scan.

Both *InputFormats* extend *MapredParquetInputFormat* and *VectorizedParquetRecordReader* so all optimizations done for reading parquet files still apply. Presto and SparkSQL work out of the box on the Hive metastore tables, provided the required *hoodie-hadoop-mr* library is in classpath.

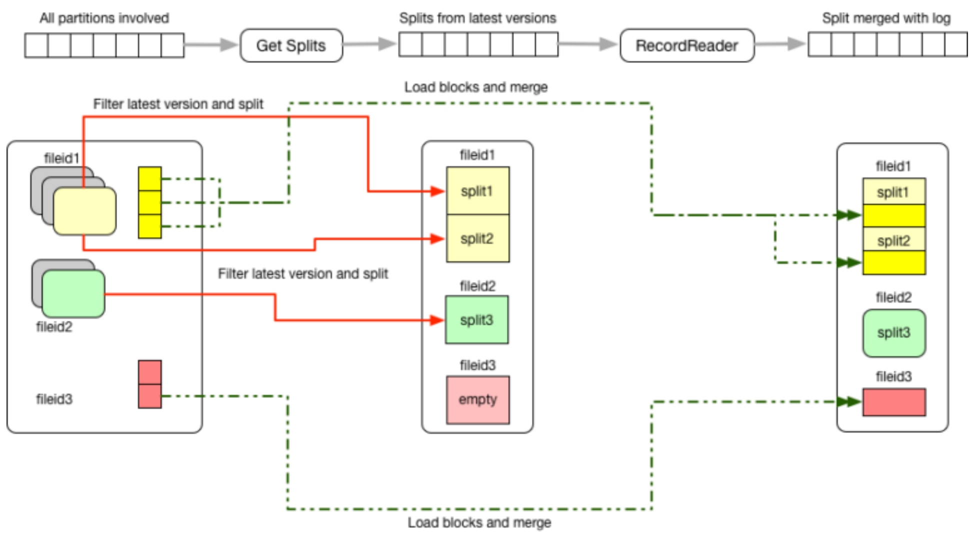

Figure 6: Hudi datasets filter the latest versions and merges them with the log before serving records.

Incremental Processing

As previously stated, modeled tables need to be processed and served in HDFS for HDFS to become the unified serving layer. Building low-latency modeled tables requires the ability to chain incremental processing of HDFS datasets. Since Hudi maintains metadata about commit times and file versions created for every commit, incremental changeset can be pulled from a Hudi-specific dataset within a start timestamp and an end timestamp.

This process works much in the same way as a normal query, except that the specific file versions that fall within the query time range are picked instead of just the latest version, and an additional predicate about the commit time is pushed onto the file scan to retrieve only the records that changed in the requested duration. The duration for which changesets can be obtained is determined by how many versions of data files can be left uncleaned.

This enables stream-to-stream joins with watermarks and stream-to-dataset joins to compute and upsert modeled tables in HDFS.

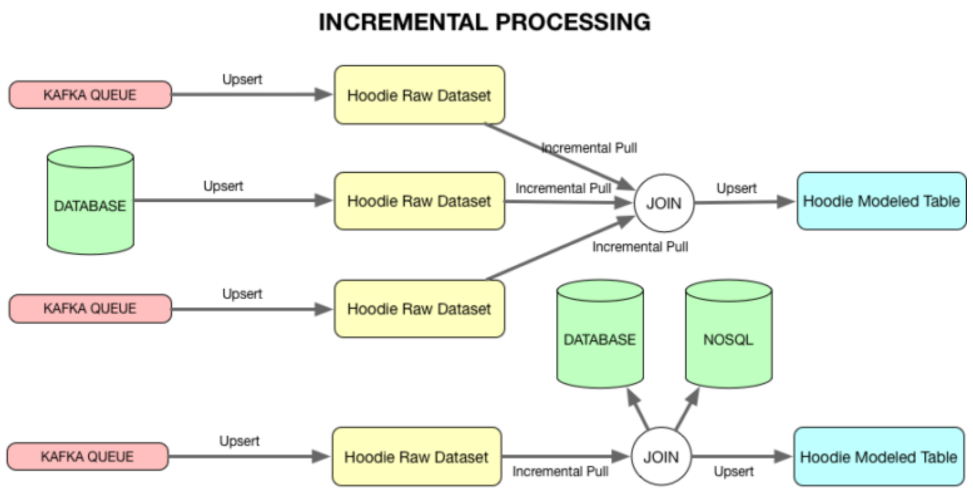

Figure 7: Hudi enables chaining computations so modeled tables can be served in Hadoop.

What's in Store Next?

Most of the technology described in this article refers to the current generation (termed as Merge-on-Read) of Hudi, which is still under active development. In the coming months, Hudi is scheduled to replace the previous generation (termed as Copy-on-Write) of storage used at Uber. The previous generation simplifies the architecture by eliminating the log files and taking a hit on latency. This has been powering Uber's data ingestion and modeled tables for several months now.

As Hudi continues to push the boundaries on latency to make ingestion in HDFS faster, there will inevitably be some iterations on identifying bottlenecks as we scale out. A few of the potential bottlenecks we intend to work on relate to speeding up indexing with an embedded global immutable index and designing a custom indexable log storage format to optimize with disk seeks on merge. Thus, we welcome feedback and encourage you to make contributions to our project.

Editor's note: *The original version of this article referred to Hudi by its nickname, Hoodie. We have updated the article to use the name of its open source project, Hudi, to avoid confusion.*

Prasanna Rajaperumal is a senior software engineer on the Data Infrastructure team, and Vinoth Chandar is a staff software engineer on the Mobile Platform team. Both are based in Uber's San Francisco engineering offices.

Our team will discuss Hudi at the upcoming Strata+Hadoop World Conference on Thursday March 16, 2017 in San Jose, CA; visit our booth for a working demonstration. If you need to contact us offline, please use hoodie mailing list or email us at hoodie-users@googlegroups.com.

Photo Header Credit: "Elephant spraying water in early evening" by Conor Myhrvold, Okavango Delta, Botswana.