Quiz 01

Everything about Big Data

Vũ Lê Thế Anh (20C13002)

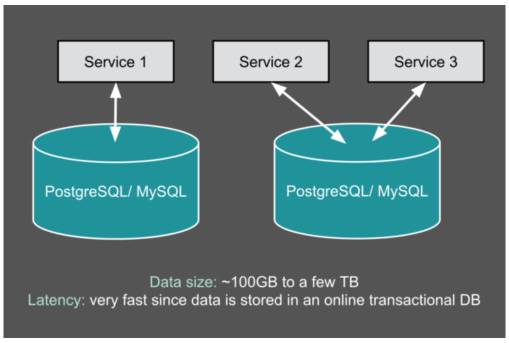
Part A – Big Data application in practice

Uber, or its full name **Uber Technologies, Inc.**, is an American company specializing in providing a smartphone-based platform to connect users in need of ride-booking or food-delivering with corresponding drivers. It was founded on 2009 by Garrett Camp and Travis Kalanick solely for San Francisco. Uber has since expanded its scope of operation to almost all continents and many countries, with about 100 million users active monthly world-wide [1] (before the pandemic, this figure reached more than 110 million). With the fast growing rate, it is expected that Uber needs to utilize Big Data frameworks in its ecosystem, and it certainly does so. A 3-year old blog on Uber Engineering [2] provides in detail the switch from traditional to big data framework and a recent blog on the same site [3] demonstrates a big data framework developed by Uber since 2017 [4].

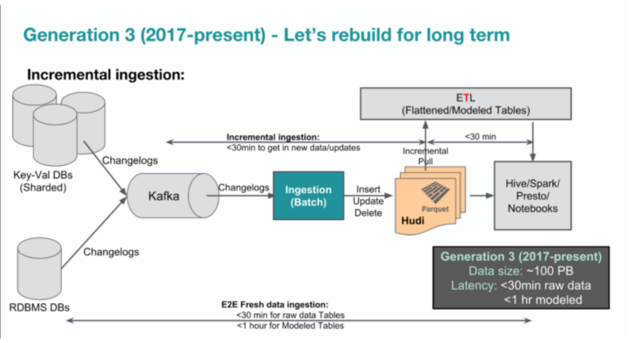
The core commitment of Uber is to provide safe and reliable transportation. To do so, it needs to process and analyze multiple types of data, from the number of bookings/orders and available drivers, to their locations and the state of traffic. Often times, it needs to make real-time predictions and thus decisions, for example, forecasting rider demand in extreme events [5]. This calls for a pipeline of data cleaning, storing, and serving with minimal latency. The pipeline needs to be reliable (fault-tolerant, able to restore from errors, etc.), scalable (handle increasing amount of data), easy to use, and with high efficiency.

According to [2], before 2014, the limited data of Uber could fit into traditional framework like MySQL or PostgreSQL. Its data was spread across multiple databases without a global management. In other words, developers at Uber needed to have adequate knowledge of where to get the data they need and write their own implementation to aggregate these data. This was manageable at the current scale of data but not for long with the exponential growth of the business. With the introduction of new locations, new drivers, new users, the need to have a global view of the data and to gain access to all data from one place rises.

By 2017, Uber has already adapted multiple big data frameworks like Hadoop, Spark, Hive, Presto, etc. to provide their engineers and analysts ease of access to the global database. From the scale of only few terabytes (TB) in 2014 at most, the data grows to the hundred of petabytes in 2017 [2]. The blog details that at the time, there are over 100 petabytes of data in HDFS, 100 thousands vcores (virtual cores configured by YARN to share usage of a physical CPU in a Hadoop cluster), 100 thousand Presto queries per day, 10 thousands Spark jobs per day, 20 thousands Hive queries per day. In a blog introducing Uber's Data Lake in June 2020 [3], it is stated that Uber ingests more than 500 billion records per day into the 150 petabytes data lake, with over 10 thousand tables and thousand of data pipelines requiring 30 thousands vcores per day, serving 1 million queries per week across multiple services. Another example of the scale of data processed at Uber is demonstrated in the blog post introducing Gairos in early 2021 [6]. In this blog, Uber says that Gairos has more than 1500 TB of queryable data, totaling to 4.5 trillion records across over 20 clusters. Every second there are 1 million events flowing into Gairos for analysis. These statistics should show a hint of how large the data is under the hood of the Uber operation.


Service 1
Service 2
Service 3
PostgreSQL/ MySQL
PostgreSQL/ MySQL
Data size: ~100GB to a few TB
Latency: very fast since data is stored in an online transactional DB

(a) Before 2014


Generation 3 (2017-present) - Let's rebuild for long term
Incremental ingestion:
ETL
(Flattened/Modeled Tables)
Incremental ingestion:
<30min to get in new data/updates
<30 min
Incremental Pull
Changelogs
Key-Val DBs (Sharded)
Kafka
Changelogs
Ingestion (Batch)
Insert
Update
Delete
Parquet
Hudi
Hive/Spark/ Presto/ Notebooks
Changelogs
RDBMS DBs
Generation 3 (2017-present)
Data size: ~100 PB
Latency: <30min raw data
<1 hr modeled
E2E Fresh data ingestion:
<30 min for raw data Tables
<1 hour for Modeled Tables

(b) After 2017

Figure 1: Differences between data management at Uber before and after introduction of Big Data framework. [2]

A key technology of Uber is its development of its own big data processing framework, called Hudi (pronounced "hoodie", short for Hadoop Upsert Delete and Incremental) [4]. In Lambda architecture, there are a batch layer to process batch data every fixed interval and a streaming layer to immediately produce an approximation state which is corrected by the hours-late batch layer state. This requires maintaining two different states at the same time, which is cumbersome. The Kappa architecture proposes the removal of the batch layer and instead let the stream processing handle the full problem. This also has two serving layers, one for the historic data, and one for the data retained (usually data in a short period with respect to the full data). Hudi is built for the purpose of unifying the serving layers, remedy the problems of the two-layer architecture above. To do so, it supports quick mutation of HDFS datasets, optimized analytical scan (for example, compute statistics of a column), and efficient update chaining and propagation. Using Hudi, Uber built one of the largest transactional data lakes in the world [3].

Another of Uber's key technology is Gairos [6], its real-time data processing, storage, and query platform. According to its introduction blog, it allows users to query data at high level, allows developers to experiment without affect the users, and it is optimized to have high throughput and low latency. Gairos is built as an utility for the application to make informed decision. For example, it provides a service to read current demand (bookings/orders) and supply (available drivers) data to calculate the surge multiplier at a location and time.

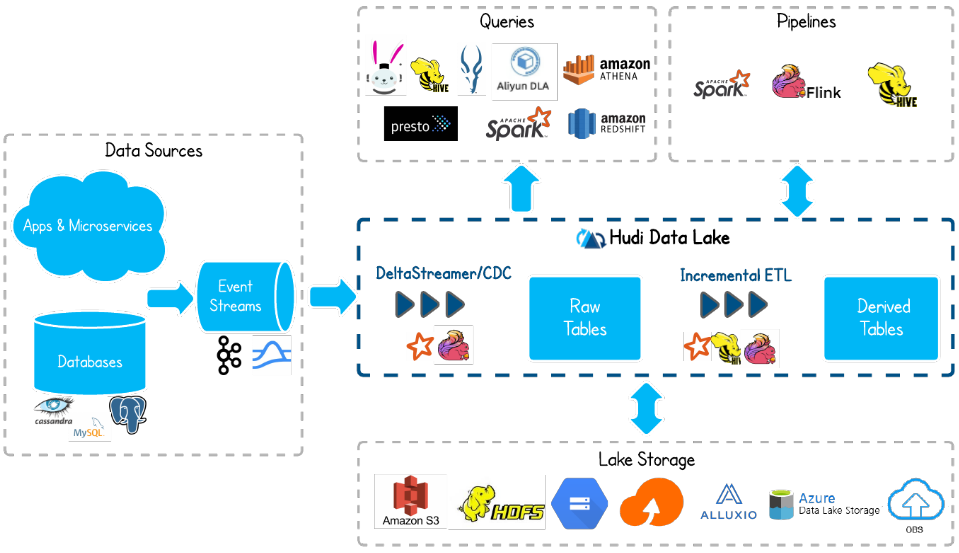


Figure 2: Apache Hudi Data Lake

Part B – Data processing framework in Big Data

Apache Flink is an open-source data processing framework which aims to unify stream and batch processing frameworks. It was developed by the Apache Software Foundation and released in May 2011. Its Github repository can be found here.

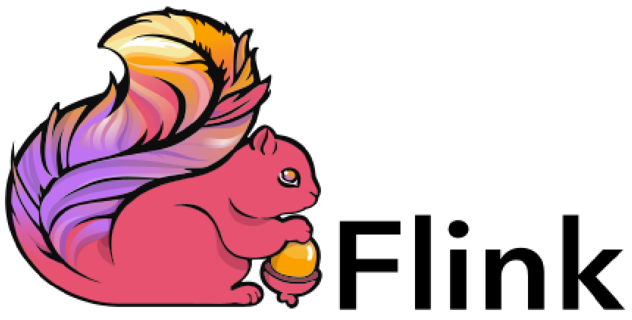


Figure 3: Logo of Apache Flink

Flink supports both batch and stream processing. It is written in Java and Scala and supports multiple languages. Programs can be written in Java, Scala, Python, and SQL. It has its own file system abstraction but provides integration with other filesystems such as Hadoop HDFS, Amazon S3, Google Cloud Storage, etc. through a bridge to Apache Hadoop [7]. Flink can integrate with common cluster resource managers like Hadoop's YARN, Apache Mesos, Kubernetes [8]. It is an alternative to Hadoop MapReduce, but provides a package to wrap functions implemented for MapReduce to embed in Flink programs [9].

One major component of Flink are the APIs provided in multple levels of abstraction [10].

- At the lowest level, the stateful and timely stream processing API allows user to process events from one or more streams with consistent, fault tolerant state. Stateful stream processing stores information across multiple events to process coming events, and

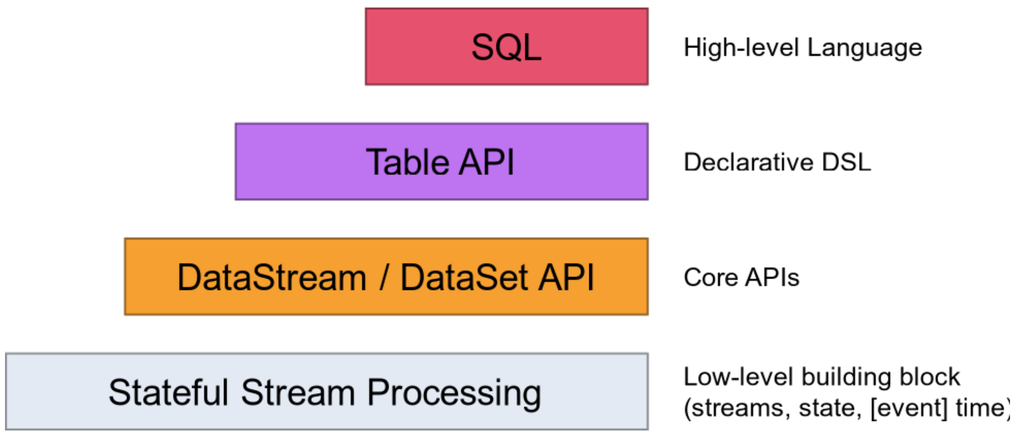


Figure 4: Flink's APIs in different levels of abstraction [10]

timely takes time into account when aggregating (for example, time series analysis).

- The Core APIs consist of the DataStream API for bounded and unbounded streams and the DataSet API for bounded data sets. This provides users with methods to transform the data for either form (DataSet for static, offlline processing and DataStream for streaming, online processing).
- The Table API views data as tables similar to relational databases and offers operations similar to SQL: select, join, group by, etc. Users can convert between tables and DataStream/DataSet.
- At the highest levele of abstraction is the SQL API. This offers user the ability to write programs as SQL query expressions, and the queries can be executed over tables defined in the Table API.

Based on the article [11], I summarize some key differences (with reference) in functionalities of Flink vs Hadoop/Spark:

- Hadoop is built for batch processing. Although Spark is said to have been built for stream processing, under the hood, its mechanism is to process data in mini-batches (collection of events over a period), so it is only near-real time. Flink provides a unified framework for both batch and stream processing. It does event-level processing (or real-time streaming). With Flink, a batch is a bounded stream.
- Same with Hadoop and Spark, Flink has a memory management system. It is an automatic system implemented separately from Java's garbage collector. This helps Flink avoid out of memory errors in a more flexible way and reduce garbage collection overhead. [12]
- Same with Hadoop and Spark, Flink has a fault-tolerant mechanism. The mechanism is based on the Chandy-Lamport distributed snapshots. It draws consistent snapshots of the distributed data stream and operator state to which the system can fall back in case of a failure. This helps maintain high throughput and provide strong consistency guarantees. This checkpointing mechanism also ensures that Flink processes every record exactly once (but sometimes, only at least once), while Hadoop does not guarantee duplication elimination. [13]

- Flink supports iterative processing unlike Hadoop and Spark (iterates in batches). It iterates naturally through its streaming architecture. [14]
- Flink has an optimizer working independent of the programming interface, unlike Hadoop and Spark (where users have to optimize the jobs manually, although there may be extensions to assist in doing so). The optimizer works similarly to optimizers in relational databases.
- Flink offers a web interface for submitting and executing jobs. This is similar to Hadoop (with zoomdata) and the same as Spark (through Zeppelin). [15]
- For security, Flink uses Kerberos Authentication, same as Hadoop. [16]
- Flink uses DataSet as abstraction for batch and DataStream for stream. For Spark, it is RDD for stream. For Hadoop, there is no abstraction. [10]
- Flink has an integrated interactive shell written in scala, same as Spark. Hadoop does not have any. [17]
- Flink has its own Scheduler but can use Hadoop's YARN scheduler. [18]
- Flink has the Table API which supports writing SQL queries as programs. Spark has a similar functionality with Spark-SQL while Hadoop uses Apache Hive. [10]
- Flink has the FlinkML library for Machine Learning with controlled cyclic dependecy graph in runtime (although has been deprecated in favor of building a new library [19]). Spark also has its own set of machine learning library while Hadoop depends on Apache Mahout.

References

[1] "Uber's users of ride-sharing services worldwide 2017-2020," Feb 2021. [Online]. Available: https://www.statista.com/statistics/833743/us-users-ride-sharing-services/

[2] R. Shiftehfar, "Uber's big data platform: 100+ petabytes with minute latency," Apr 2020. [Online]. Available: https://eng.uber.com/uber-big-data-platform/

[3] N. Agarwal, "Building a large-scale transactional data lake at uber using apache hudi," Jun 2020. [Online]. Available: https://eng.uber.com/apache-hudi-graduation/

[4] P. Rajaperumal, "Hudi: Uber engineering's incremental processing framework on apache hadoop," Dec 2018. [Online]. Available: https://eng.uber.com/hoodie/

[5] N. Laptev, "Engineering extreme event forecasting at uber with recurrent neural networks," Jan 2019. [Online]. Available: https://eng.uber.com/neural-networks

[6] G. Zhao, "Uber's real-time data intelligence platform at scale: Improving gairos scalability/reliability," Jan 2021. [Online]. Available: https://eng.uber.com/gairos-scalability/

[7] "Flink file systems." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/internals/filesystems/

[8] "Flink architecture." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/concepts/flink-architecture/

[9] "Hadoop compatibility in flink." [Online]. Available: https://flink.apache.org/news/2014/11/18/hadoop-compatibility.html

[10] "Concept overview." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/concepts/overview/

[11] "Hadoop vs spark vs flink – big data frameworks comparison," May 2021. [Online]. Available: https://data-flair.training/blogs/hadoop-vs-spark-vs-flink/

[12] "Memory management improvements with apache flink 1.10." [Online]. Available: https://flink.apache.org/news/2020/04/21/memory-management-improvements-flink-1.10.html

[13] "Flink fault tolerance." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/learn-flink/fault_tolerance/

[14] "Flink dataset iterations." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/dev/dataset/iterations/

[15] "Flink on zeppelin notebooks for interactive data analysis - part 1." [Online]. Available: https://flink.apache.org/news/2020/06/15/flink-on-zeppelin-part1.html

[16] "Flink with kerberos authentication." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/deployment/security/security-kerberos/

[17] "Scala repl." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-master/docs/deployment/repls/scala_shell/

[18] "Flink's job scheduling." [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/internals/job_scheduling/

[19] "[flink-12470] flip39: Flink ml pipeline and ml libs - asf jira." [Online]. Available: https://issues.apache.org/jira/browse/FLINK-12470