HELLENIC REPUBLIC
UNIVERSITY OF IOANNINA

Uber & Big Data
a case study

Christos Gogos

11/5/2020

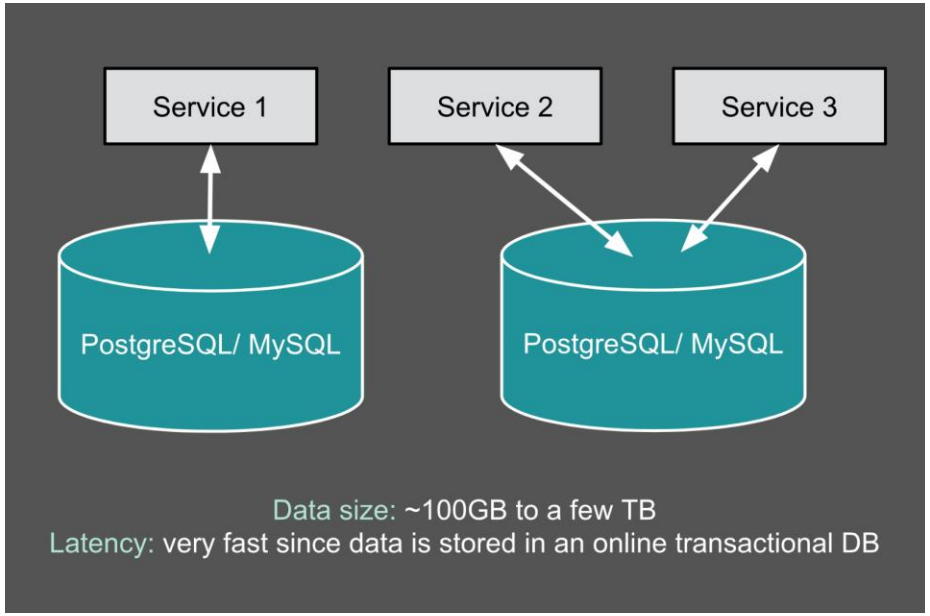
https://github.com/chgogos/big_data

Uber

- Founded at 2009 by Travis Kalanick and Garrett Camp
- Peer to peer ridesharing, taxi cab, food delivery, bicycle sharing
- Uber's services and mobile app officially launched in San Francisco in 2011
- Operations in 785 metropolitan areas worldwide (Sept. 2018)
  - 12000+ employees

Petabytes

- Uber relies heavily on making data-driven decisions at every level
  - Forecasting rider demand during high traffic events
  - Addressing bottlenecks in driver-partner signup process
- Need for store, clean and serve over 100 Petabytes of data (2017) with minimum latency.
- Need for a big data solution:
  - Reliable
  - Scalable
  - Easy to use
  - Fast
  - Efficient



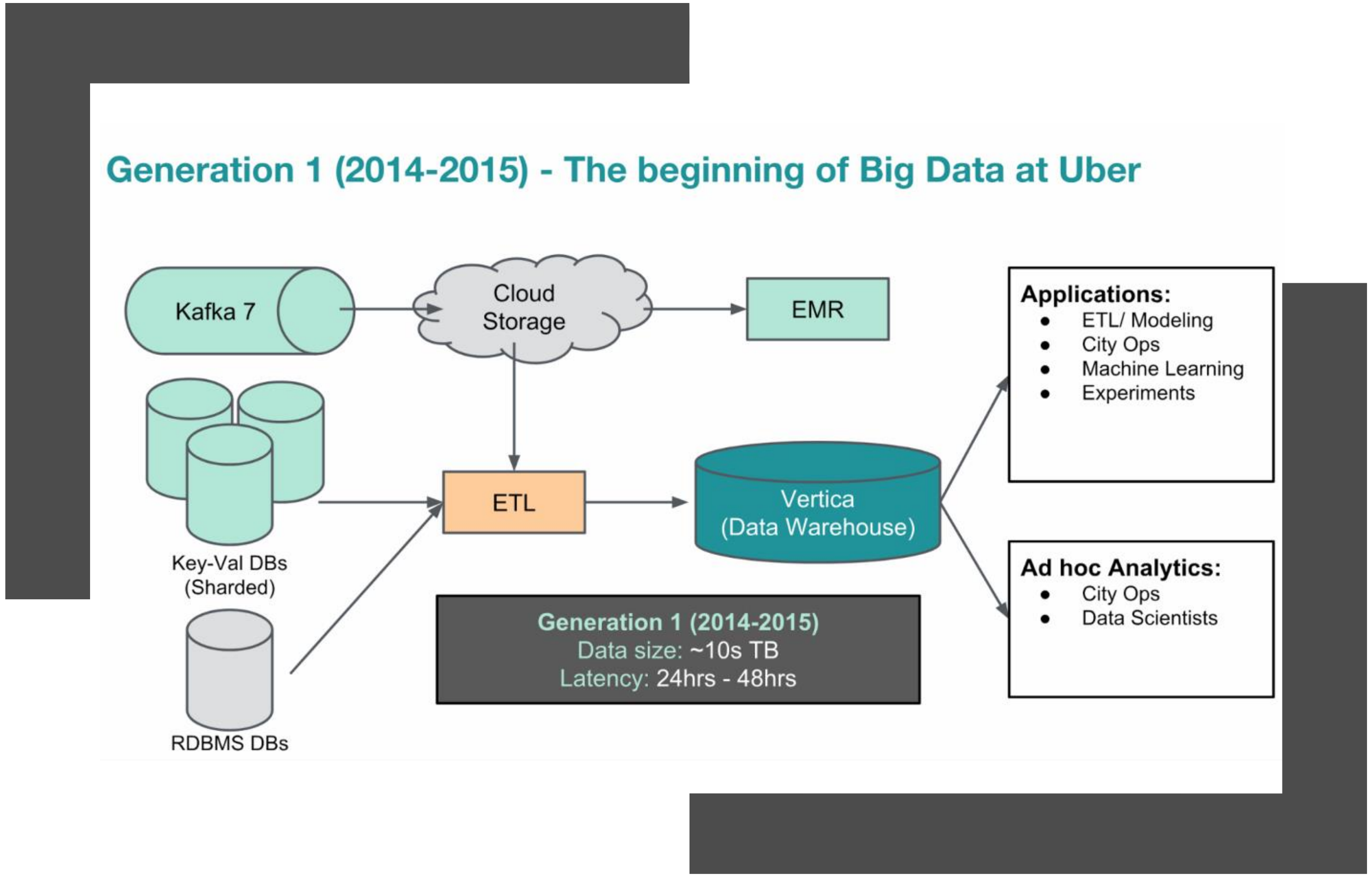
Generation 0 (prior to 2014)

- data size = few terabytes
- latency < 1 min
- Online Transaction Processing (OLTP) databases
  - MySQL
  - PostgreSQL
- No global view of all stored data
- Soon, the exponential growth of the company led the build of an analytical data warehouse

Data users

- **City operations teams (thousands of users)**
  On-the-ground crews that manage and scale Uber's transportation network in each market. Access data on a regular basis to respond to driver-and-rider-specific issues
- **Data scientists and analysts (hundreds of users)**
  Analysts and scientists spread across different functional groups that need data to help deliver high level transportation and delivery experiences to the users (e.g. forecasting rider demand)
- **Engineering teams (hundreds of users)**
  Engineers focused on building automated data applications, such as Fraud Detection and Driver Onboarding platforms

Generation 1 (2014-2015)



- Vertica: data warehouse software (column oriented)
- Extract Transform Load (ETL)
  - AWS S3 → Vertica
  - OLTP databases → Vertica
  - Logs → Vertica
  - …
- Online query system using SQL (city operators could easily interact with the data without knowing about the underlying technologies)
- Global view of data was achieved

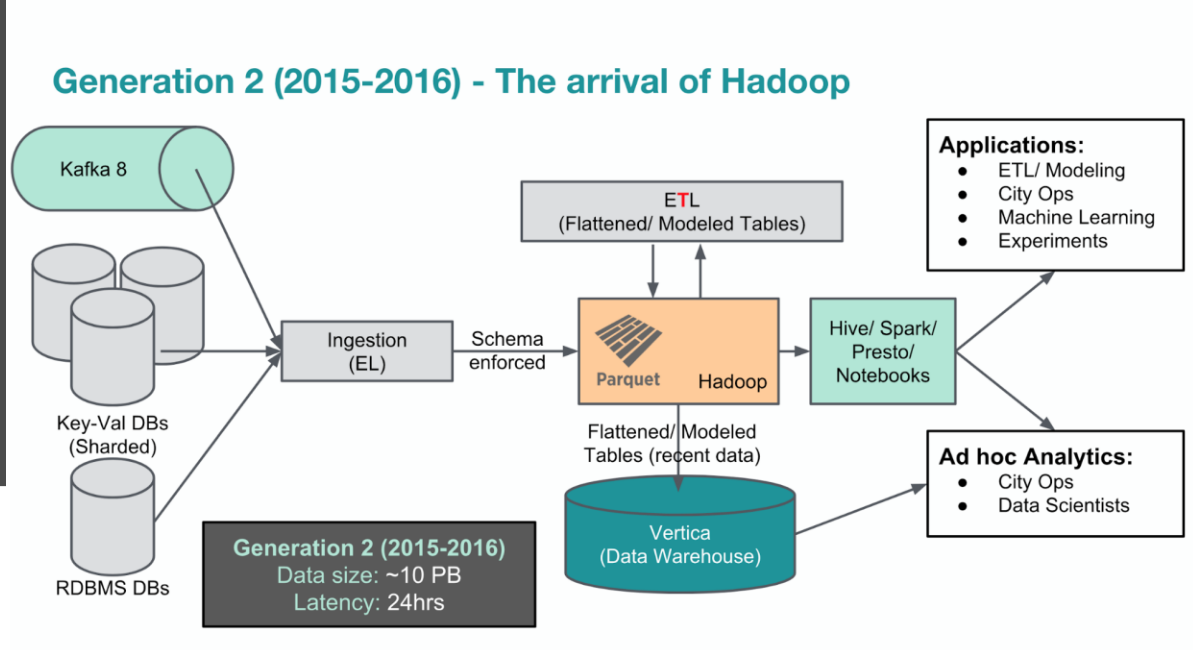
Generation 1 (2014-2015)

- Data Size = 10s of terabytes
- # users = several hundreds

Limitations of Generation 1

- Data (in JSON format) was ingested through ad hoc ETL jobs → data reliability became an issue
- Lack of a formal schema communication mechanism → duplicate data
- Expensive scaling

Generation 2 (2015-2016)

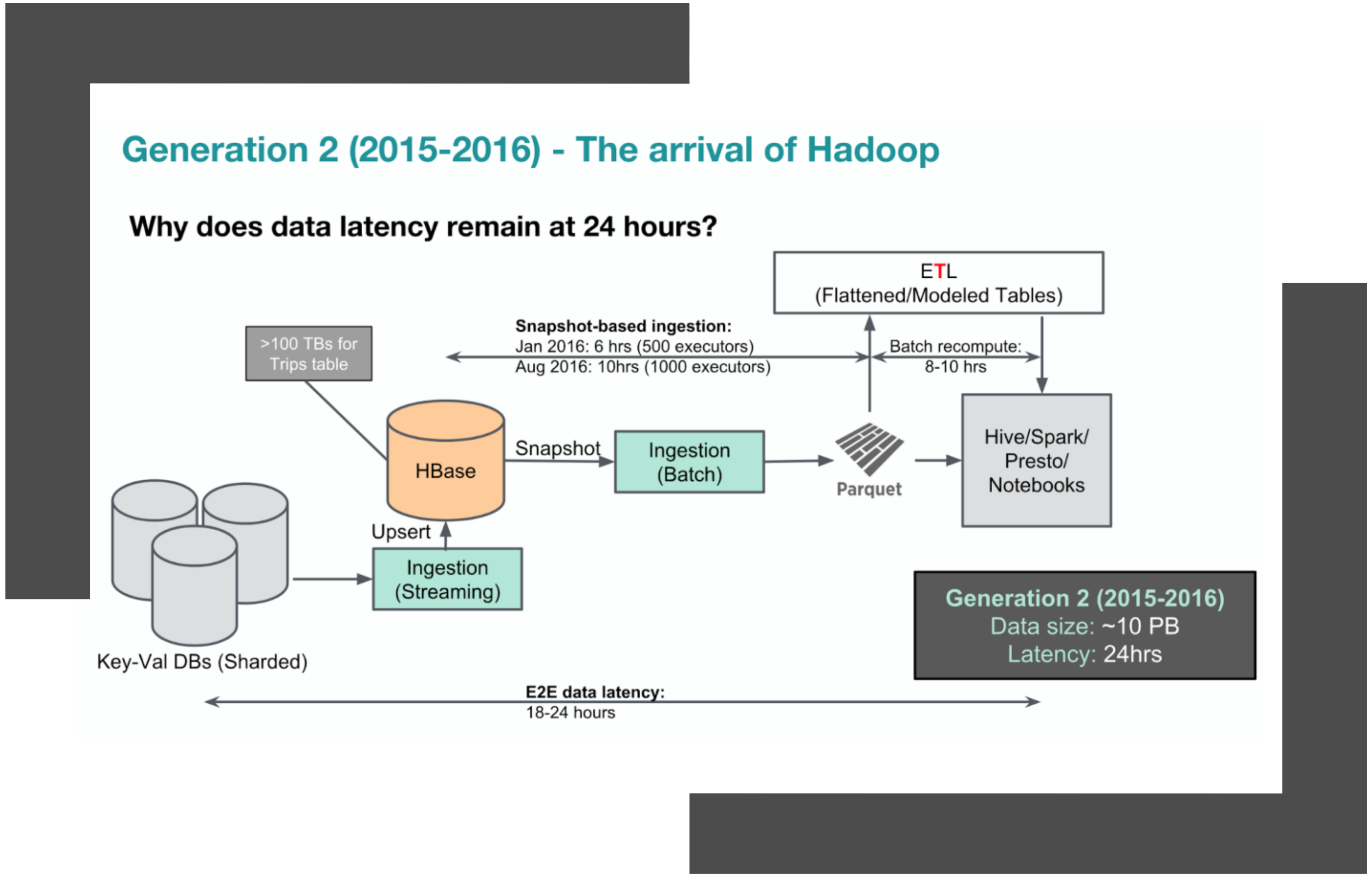


- **Hadoop data lake** (all raw data was ingested from different online data stores only once and with <u>no transformation during ingestion</u>)
- Access data
  - Presto: interactive ad hoc user queries
  - Apache Spark: programmatic access to raw data
  - Apache Hive: heavy queries
- All data modeling and transformation only happened in Hadoop
- Critical tables were transferred to the data warehouse
  - quick SQL queries
  - lower operational cost
- Transition from JSON to Apache Parquet
  - higher compression
  - integration with Apache Spark

Generation 2 (2015-2016)

- Data Size = 10s of petabytes
- Data platform = 10,000 vcores, 100,000 running batch jobs / day
- # users = thousands

Limitations of Generation 2



- Massive amount of small files stored in HDFS → pressure on HDFS NameNodes
- New data was accessible to users once every 24 hours → no real-time decisions
- 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
    - swap the output tables
    - view the new data

Pain points in gen2, solutions adopted in gen3

- **HDFS scalability limitation:** HDFS is bottlenecked by its NameNode capacity (if data size > 50-100 PB)

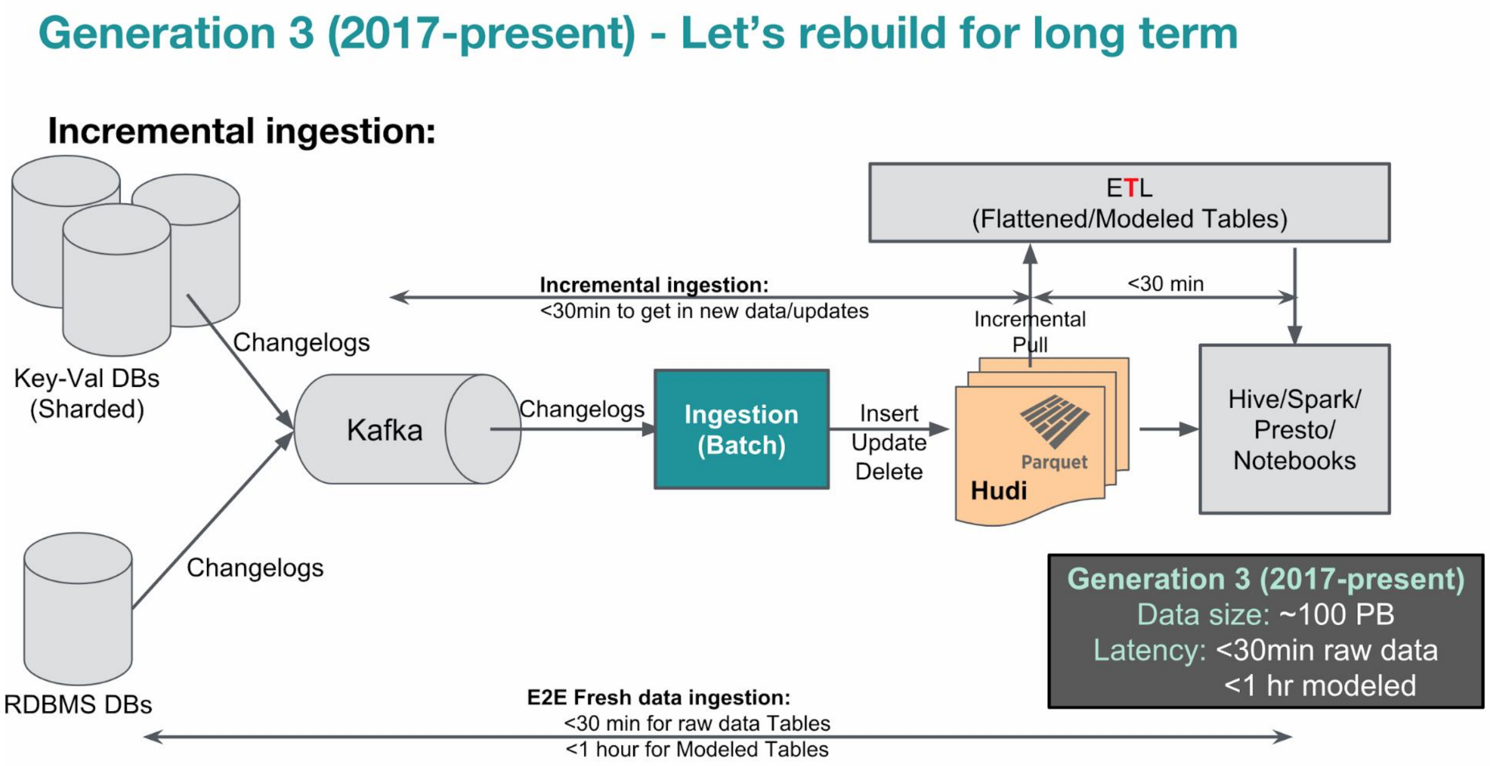
  Solution: control number of small files, move data to separate clusters
- **Faster data in Hadoop:** 24-hr data latency
  - **Solution:** incremental ingestion of only updated and new data
- **Support of updates and deletes in Hadoop and Parquet:** ingest all updates at one time, once per day
  - **Solution:** framework to support update/delete operations over HDFS
- **Faster ETL and modeling:** rebuild derived tables in every run
  - **Solution:** pull out only the changed data from the raw source table, update the previous derived output table

Hudi (Hadoop Upserts anD Incremental)

- Developed by Uber engineering in order to support Generation 3
- Open source Spark library that provides an abstraction layer on top of HDFS and Parquet to support the required update and delete operations
- Allows data users to incrementally pull out only changed data
  - Data users pass on their last checkpoint timestamp and retrieve all the records that have been updated since (without scanning the entire source table)
  - Snapshot-based ingestion of raw data to an incremental ingestion model:
    data latency 24 hours → < 1 hour

Generation 3 (2017 – present)

Ingestion Spark jobs run every 10-15 minutes, providing a 30-minute raw data latency in Hadoop



Generation 3 (2017-…)

- Data Size = 100 PB data in Hadoop
- Data platform = 100,000 vcores
- ~ 100,000 Presto queries / day
- ~ 10,000 Spark jobs / day
- ~ 20,000 Hive queries / day

Generation 4 (future work)

- Improved data quality through semantic checks
- Improved data latency (5 minutes)
- New version of Hudi
  - Generate larger parquet files (1GB vs 128MB)
  - Improve management of updates on parquet files through deltas

References

- https://eng.uber.com/
- https://eng.uber.com/uber-big-data-platform/
- https://eng.uber.com/hoodie/