UBER

Uber's Data Journey: 100+ PB with Minute Latency

Reza Shiftehfar
Hadoop Platform team
reza@uber.com

04.18.2018

Who am I

Reza Shiftehfar

- PhD in Computer Science from University of Illinois @Urbana-Champaign
- with Uber since 2014
- Founding engineer of the data platform team at Uber
- Currently managing the Hadoop Platform team at Uber
- Helped scale Uber's data from a few TB to 100+ PB
- Helped lower data latency from 24+ hrs to minutes

Agenda

1. **Intro to Data @ Uber**
2. **Data Platform - Past**
   - The beginning of Big Data - Generation 1
   - The arrival of Hadoop - Generation 2
3. **Data Platform - Present**
   - Let’s rebuild for long term - Generation 3
4. **Data Platform - Future**
   - What’s coming next - Generation 4
5. **Lessons learned**

UBER

Intro to Data @ Uber:

Uber's Mission

"Transportation as reliable as running water, everywhere, for everyone"

600+ Cities

75+ Countries

And Growing...

The Impact of Data @ Uber

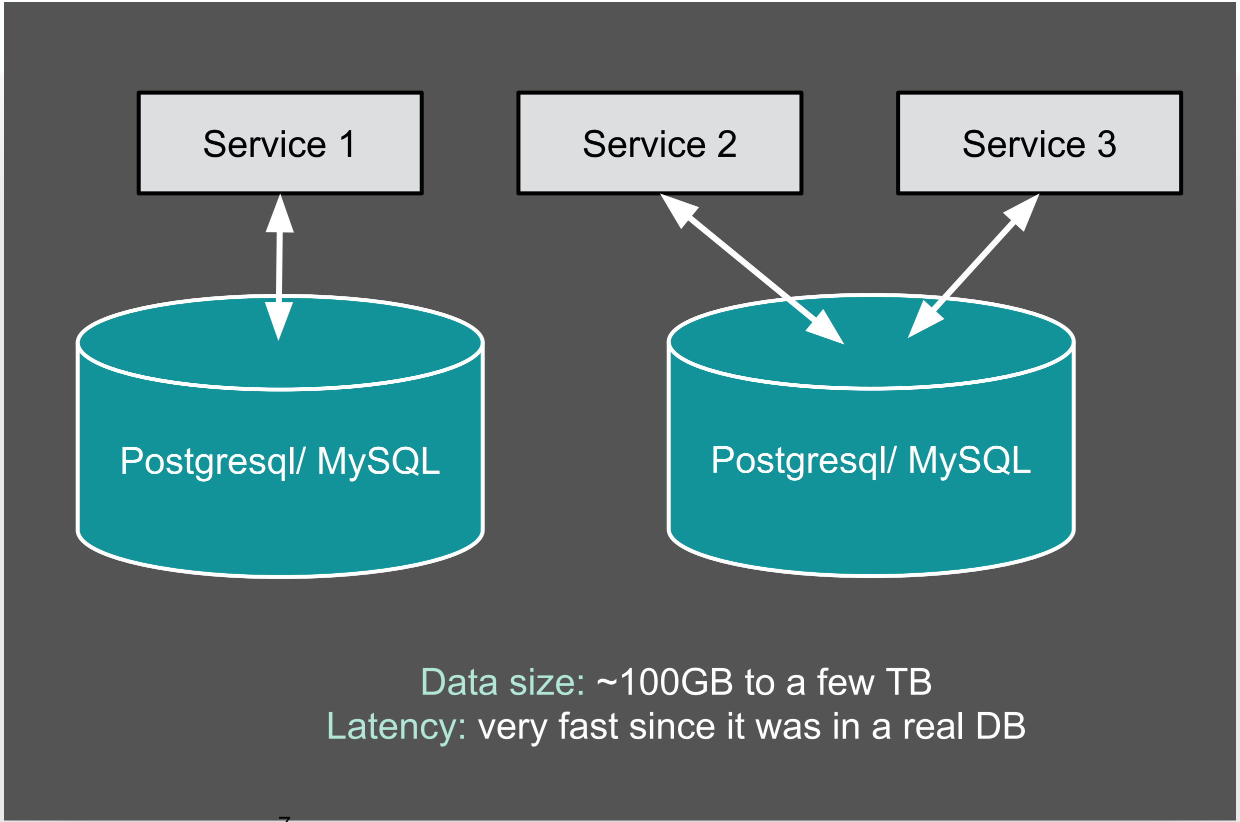
1. **City OPS (~1000s)**
    - On the ground team who run and scale uber's transportation network

2. **Data Scientists and Analysts (~100s)**
    - Spread across various functional groups (e.g. Marketing Spend, Forecasting)

3. **Engineering Teams (~100s)**
    - Focused on building automated data applications (Fraud Detection, Incentive Payments, Background Checks,...)

Not long ago (Before 2014)

- Data small enough to fit into a few OLTP DBs (MySQL/ Postgresql)
- Users had to access these DBs individually to play with the data

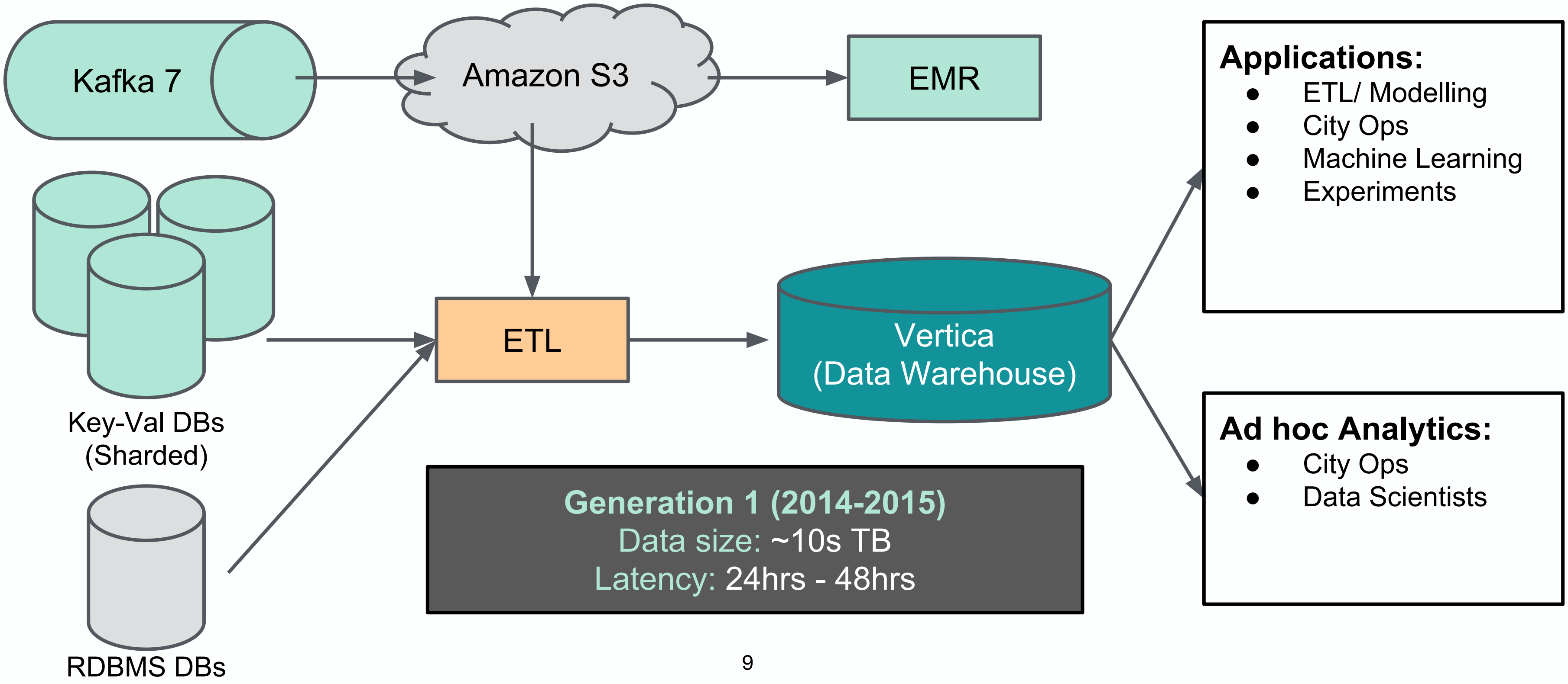


UBER

Data @ Uber: The beginning of Big Data - Generation 1

(2014-2015)

The beginning of Big Data - Generation 1 (2014-2015)



The beginning of Big Data - Generation 1 (2014-2015)

Highlights Gen. 1:

- Scalability grew to ~10s TB
- Global view of all data in one place
- Vertica support of SQL made it very popular
- More number of users could query the data in parallel (~100s)
- Applications started to build products around data (e.g. ML, Experiment,...)
- Users started to run ad hoc queries to better run the business or explore data

The beginning of Big Data - Generation 1 (2014-2015)

Problems/ Limitations:

Gen.1- Pain Point #1: Data Reliability:

- Word-of-mouth Schema communication
- Json data, breaking pipelines

Gen.1- Pain Point #2: Data Scalability:

- Exponential grow of data faster than expected
    i. Had to delete older data to free up space for new incoming data
- Many parts were not horizontally scalable (e.g. Kafka 7, Celery workers,...)
- Warehouse tool (Vertica) was used as Data Lake
    i. Raw data piling up in Vertica
    ii. Data Modelling happening in Vertica

The beginning of Big Data - Generation 1 (2014-2015)

Problems/ Limitations (cont.) :

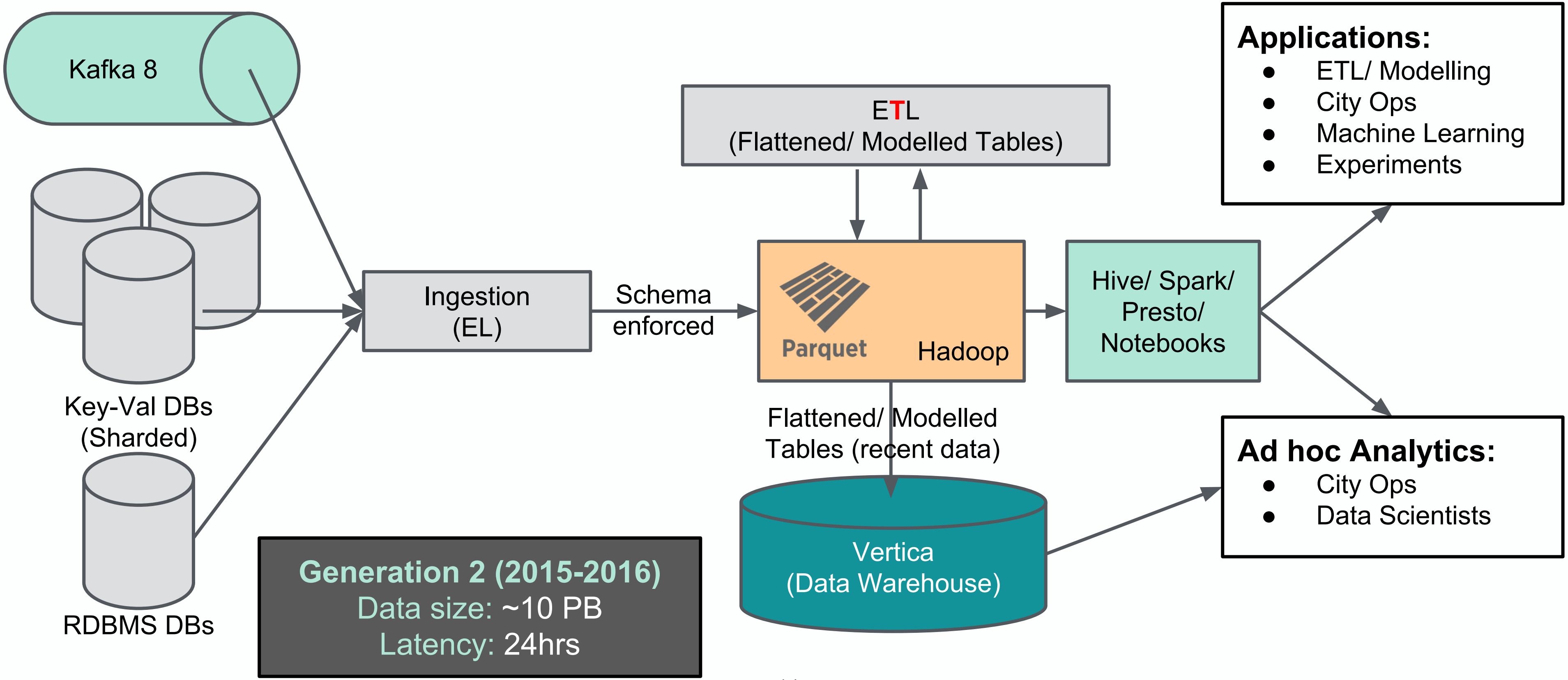
Gen.1- Pain Point #3: Fragile ingestion:

- Multiple ingestion of the same data due to Transformation in the pipeline
    1. Extra pressure on the source
    2. Multiple copies of the same data in Vertica
- ETL jobs source-dependent, stand alone jobs/scripts, hard to add new data sets/types
- Painful Backfilling because of projections & transformation in the pipelines

Data @ Uber:
The arrival of Hadoop - Generation 2

(2015-2016)

The arrival of Hadoop - Generation 2 (2015-2016)



The arrival of Hadoop - Generation 2 (2015-2016)

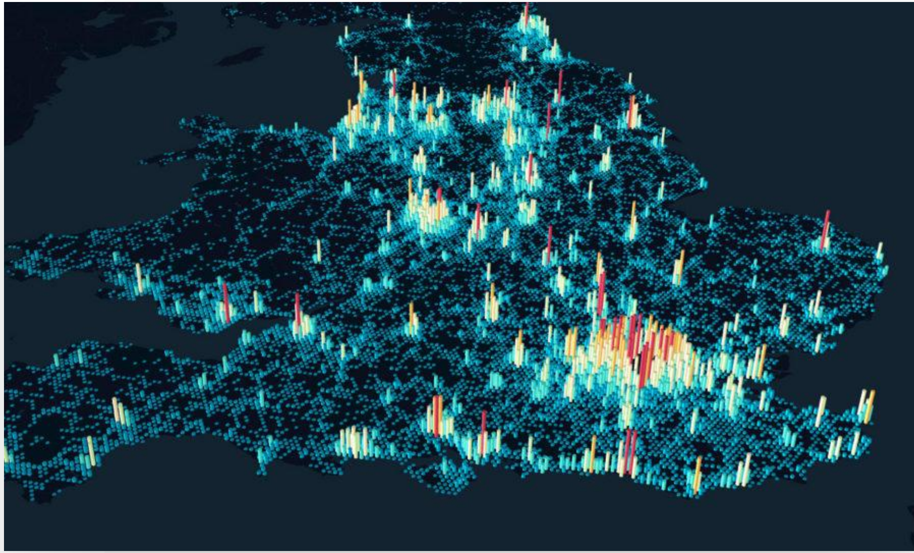
Highlights Gen. 2:

- All raw data is stored in Hadoop Data Lake
- Data stored as Columnar Parquet format
  - More efficient storage
  - More efficient queries
- All ETL/Modelling happens in Hadoop
- Subset of data transferred to warehouse
  - Only flattened selected recent dates
- Presto added as interactive query engine
- Spark notebooks added to encourage data scientists to use Hadoop

The arrival of Hadoop - Generation 2 (2015-2016)

Big Wins:

- Hadoop became the source-of-truth for all data
  - 100% of All analytical data in one place
- Hadoop powered critical Business Operations
  - Partner Incentive Payments, Fraud
- Unlocked the real power of data
- Gave us time to stabilize the infrastructure (Kafka,....) & think long-term



Some Numbers (early 2016):

- **~10 PB in HDFS**
- **~10 TB/day new data**
- **~10k vcores**
- **~100k daily batch jobs**
- **And growing...**

The arrival of Hadoop - Generation 2 (2015-2016)

Solved issues from Generation 1:

~~**Gen.1- Pain Point #1:** Data Reliability: Schema issue~~ -> Solved

- Schematized All Data (Json -> Parquet)
- Build a new central Schema-Service with client libraries for auto integration

~~**Gen.1- Pain Point #2:** Data Scalability~~ -> Solved

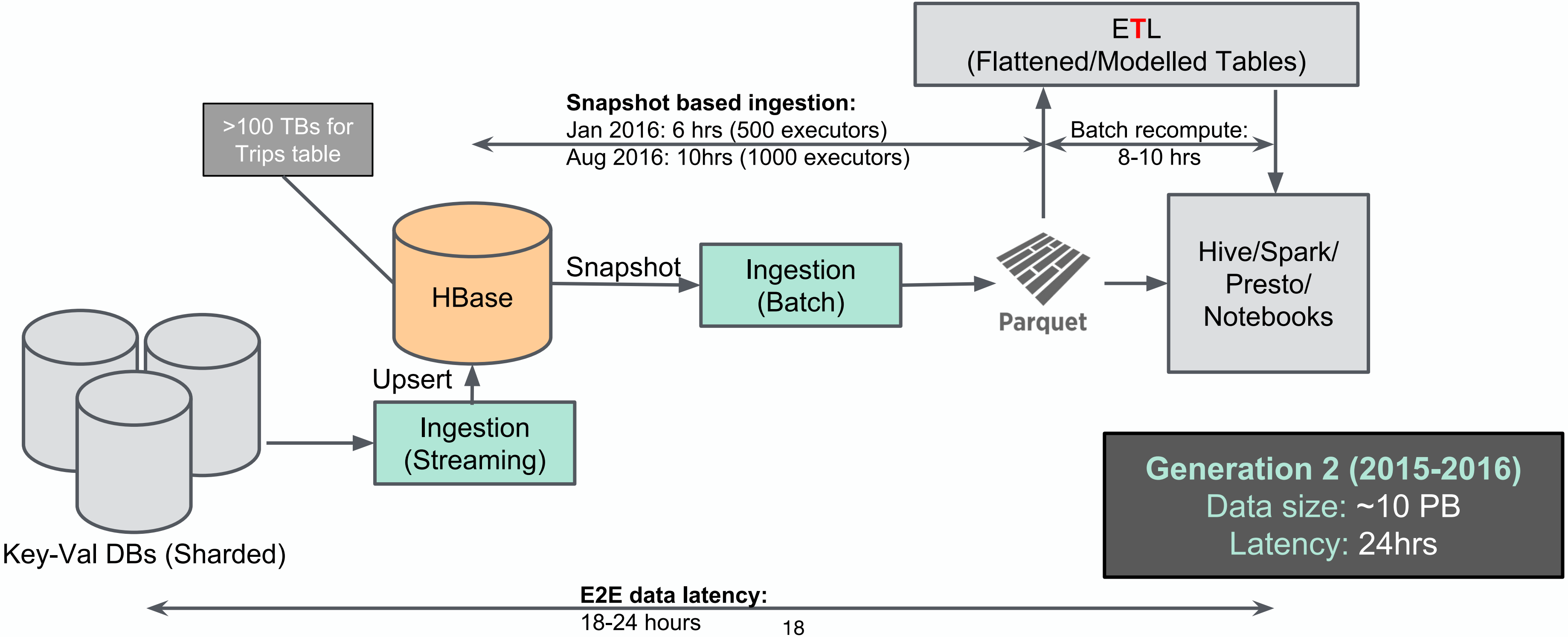
- All Infrastructure horizontally scale
- Kafka 8 & Hadoop were introduced

~~**Gen.1- Pain Point #3:** Fragile ingestion~~ -> Solved

- Hadoop Data Lake was added
    i. Store raw data in original nested format in Hadoop
- Data modelling moved to Hadoop

The arrival of Hadoop - Generation 2 (2015-2016)

Why data latency remains at 24 hours?



The arrival of Hadoop - Generation 2 (2015-2016)

Problems/ Limitations:

Gen.2- Pain Point #1: Scalability:

- Too many small files in HDFS (required async stitcher)
- Source-specific data ingestion pipelines increased maintenance cost

Gen.2- Pain Point #2: Data Latency too high:

- snapshot based ingestion results in 24hrs data latency

Gen.2- Pain Point #3: Updates became a big problem:

- Updates/late-arriving-data are natural part of our data

Gen.2- Pain Point #4: ETL/ Modelling became the bottleneck:

- ETL/Modelling was snapshot based (running daily off raw tables)

UBER

Data @ Uber:
Let's rebuild for long term - Generation 3
(2017-present)

Let’s rebuild for long term - Generation 3 (2017-present)

Some Numbers (early 2017):

- ~100+ PB in HDFS data
- ~100k vcores
- ~100k Presto queries/day
- ~1000+ Spark apps/day
- ~20k Hive queries/day
- And still growing...

Let’s rebuild for long term - Generation 3 (2017-present)

Motivation for rebuilding:

- Interactive Query engines -> Hadoop data extremely popular
- No more fire-fighting -> allowed study of our real needs

Problems to solve:

- **Gen.2- Pain Point #1:** HDFS Scalability
    - Namenode will always be the bottleneck
    - Small files are the killer
    - Benefit from ViewFS and Federation to scale
        - Controlling small files and moving part of data to a separate cluster (e.g. HBase, Yarn app logs) can let you get to 100+ PB
        - See our recent Engineering Blog post on this

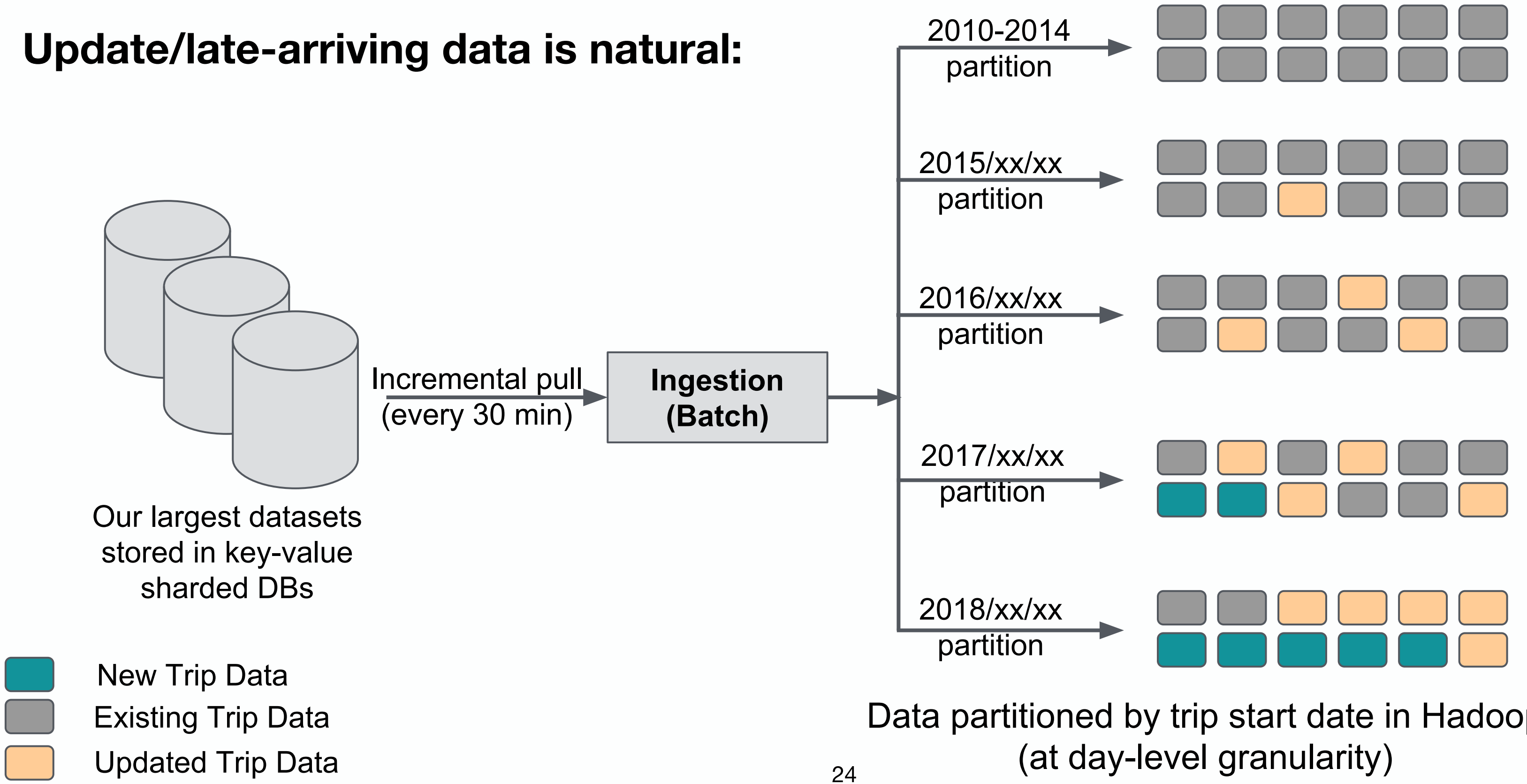
Let’s rebuild for long term - Generation 3 (2017-present)

Problems to solve:

- **Gen.2- Pain Point #2:** Faster data in Hadoop
    - Need fully incremental ingestion of data
- **Gen.2- Pain Point #3:** Support for Updates/Deletes in Hadoop/Parquet
    - Need to support Update/Deletion during ingestion of incremental changelogs
        - Out data has large number of columns with nested data support -> Parquet stays
- **Gen.2- Pain Point #4:** Faster ETL/ Modelling
    - ETL has to become incremental too
    - Need to allow users to pull out only changes incrementally
    - Have to support all different query engines (Hive, Presto, Spark,...)

Let’s rebuild for long term - Generation 3 (2017-present)

Update/late-arriving data is natural:

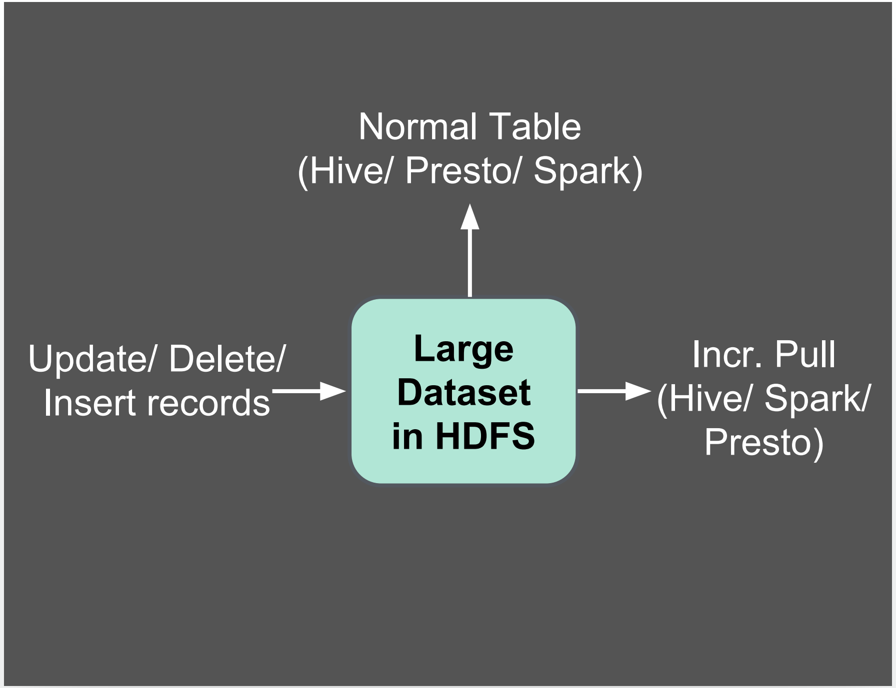


Data partitioned by trip start date in Hadoop
(at day-level granularity)

Let's rebuild for long term - Generation 3 (2017-present)

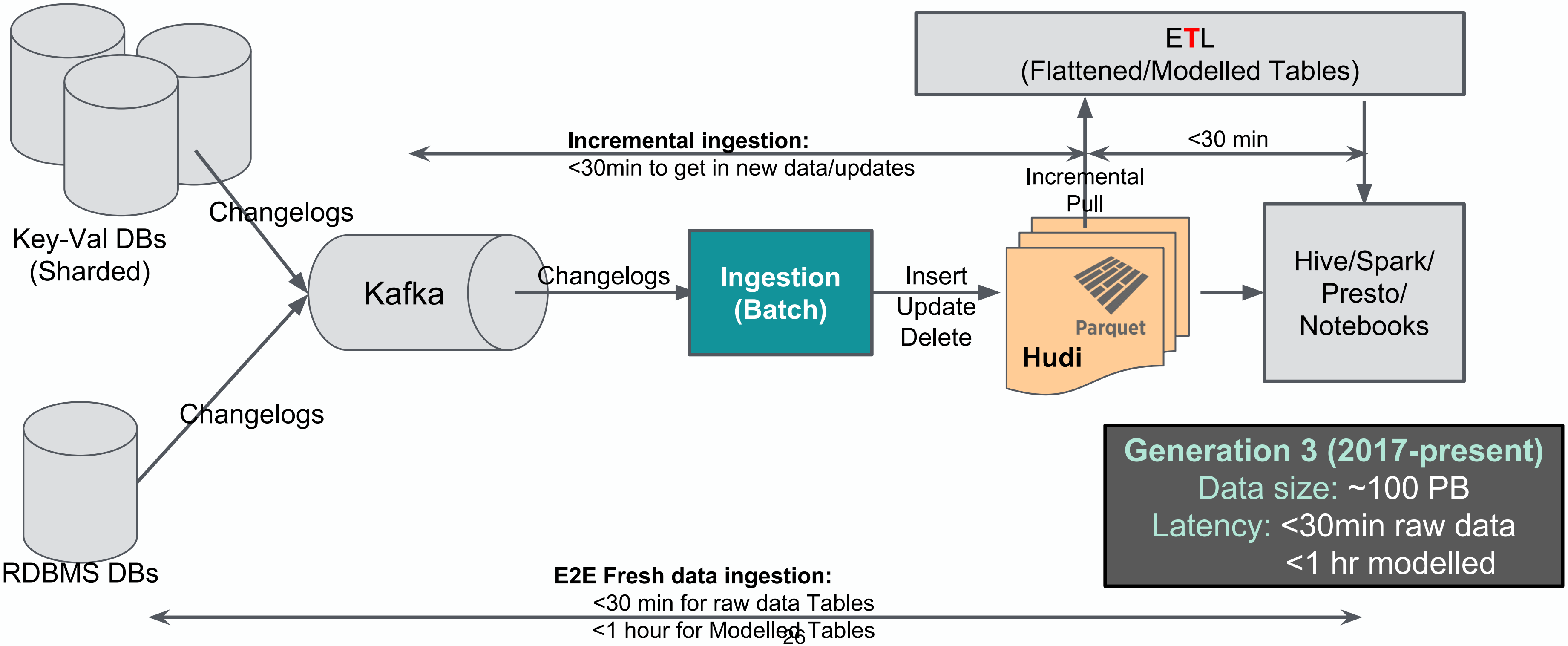
What did we build to address these needs?

- Built **Hudi**: **H**adoop **U**pserts an**D** **I**ncremental
- Storage abstraction to:
  - Apply upsert/delete on existing Parquet data in Hadoop
  - Pull out changed data incrementally
- Spark based library:
  - Scales horizontally like any Spark job
  - Only relies on HDFS
- It is open-sourced (Hudi on Github)



Let’s rebuild for long term - Generation 3 (2017-present)

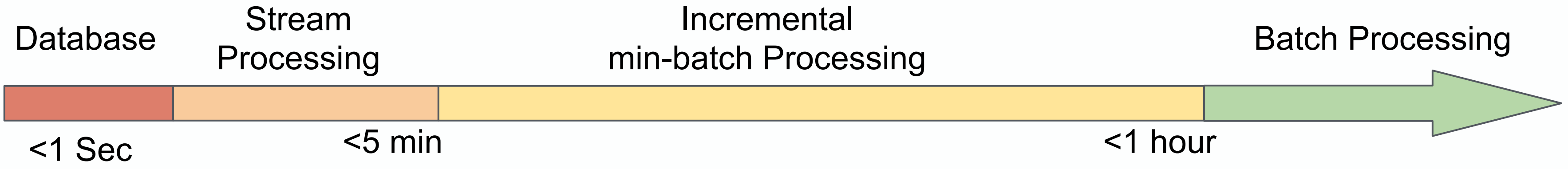
Incremental ingestion in Gen. 3:



Let’s rebuild for long term - Generation 3 (2017-present)

What is Incremental Processing:

- Traditional λ architecture provides: Streaming vs Batch solutions
  - That assumes append-only immutable data
  - Processing based on timestamp (usually skips late-arriving data)
- Incremental Processing is mini-batch jobs that pulls out only changed data
  - This gets you all the recently appended data as well as old changed/updated records
  - Provides high completeness (compared to streaming mode)
  - Processing no longer limited by updates/deletes or late-arriving data
  - Is a batch job and supports full batch functionality (e.g. joins,....)

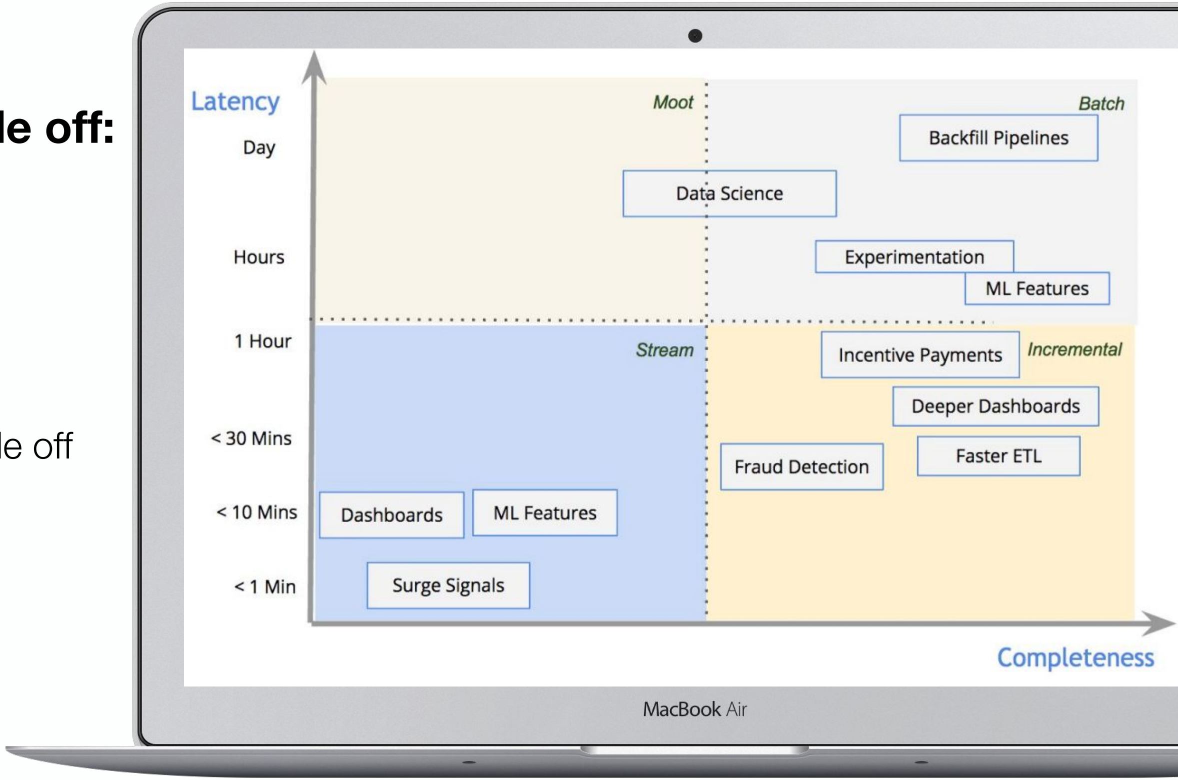


Let’s rebuild for long term - Generation 3 (2017-present)

Stream/Batch processing Trade off:

- Latency
- Completeness
- Cost (Throughput/efficiency)

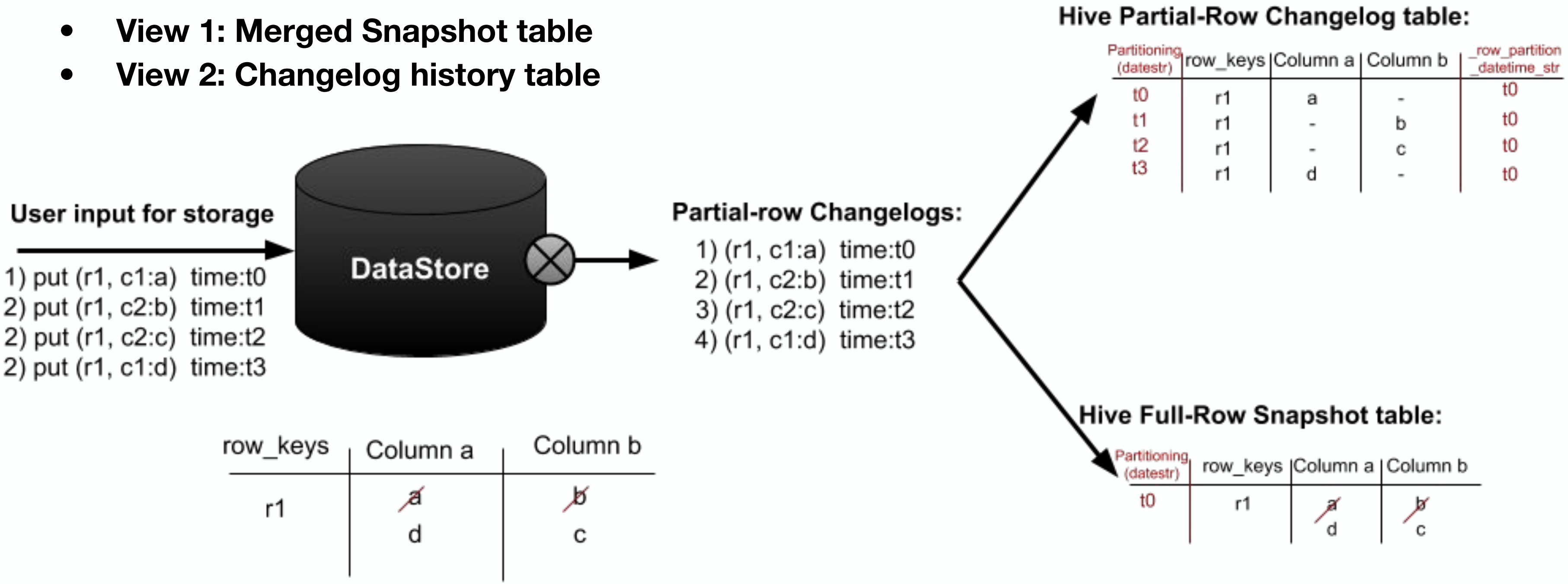
Study your use case based on these trade off



Let’s rebuild for long term - Generation 3 (2017-present)

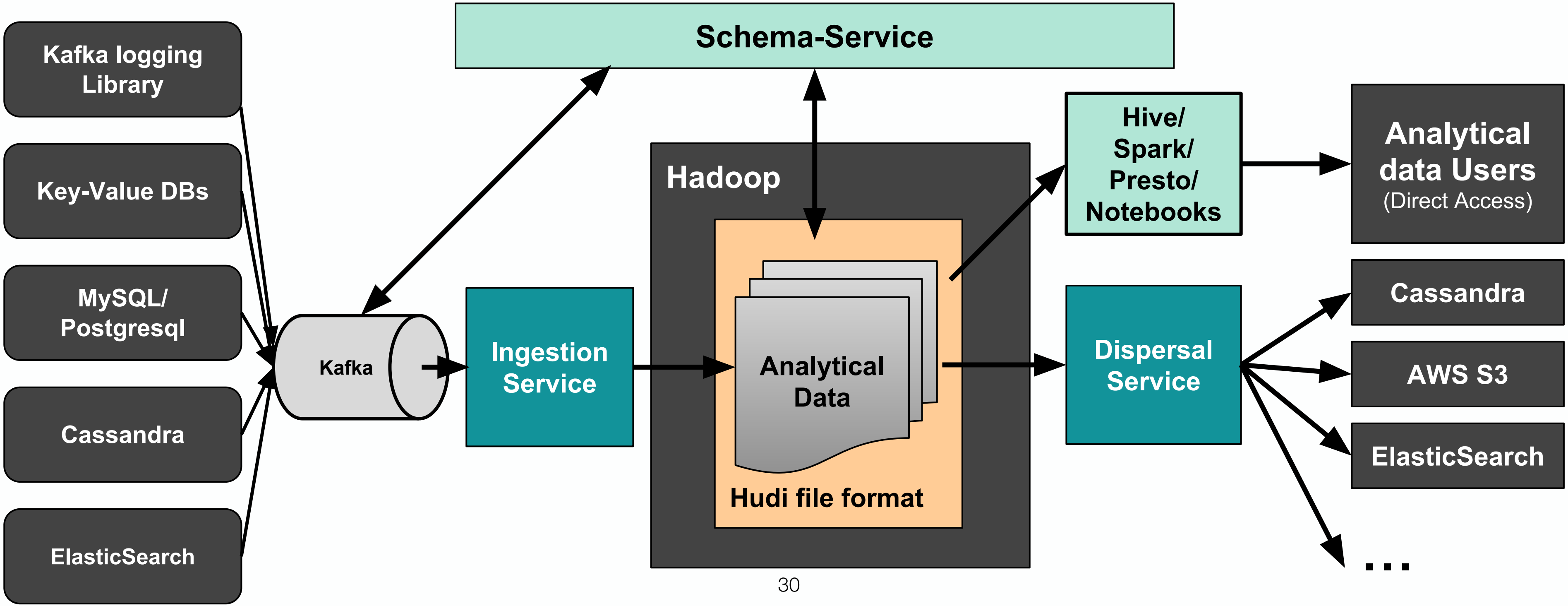
Standardized Hive raw data model:

- **View 1: Merged Snapshot table**
- **View 2: Changelog history table**



Let’s rebuild for long term - Generation 3 (2017-present)

Generic Any-to-Any Data platform (To be open-sourced soon)



UBER

Data @ Uber:
What’s coming next - Generation 4

(Ongoing effort)

What's coming next - Generation 4 (Ongoing effort)

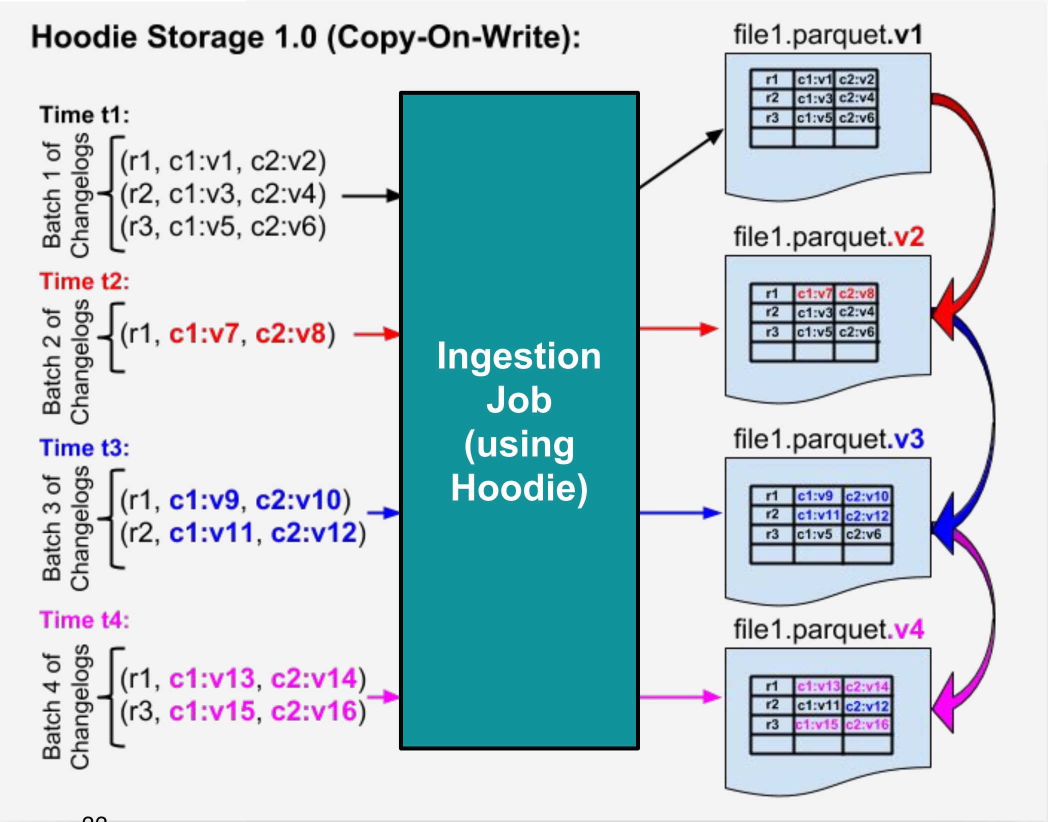
Are we done? Any remaining items?

1. **Data Quality is still a concern:**
    - Further unification of Hadoop Ingestion with strict contract with Storage team
    - Expand schema-service beyond type/structural check and into semantic checks
2. **Still Need faster data access**
    - ~5-10 min Hadoop data for mini-batching to compete with Streaming
3. **Efficiency is the next big monster**
    - Don't limit yourself to Hadoop. Go for the entire compute resources
    - Unified resource scheduler for Hadoop and beyond (Mesos, Yarn and now Peloton)
    - See our presentation at "Hadoop Infrastructure@Uber Past , Present and Future"
4. **Hoodie is still actively being developed**
    - Get rid of sensitivity with respect to the ratio of update/delete vs insert
    - Provide large Parquet file (1+ GB) with data latency of 5-10min

What's coming next - Generation 4 (Ongoing effort)

Hoodie Storage 1.0:

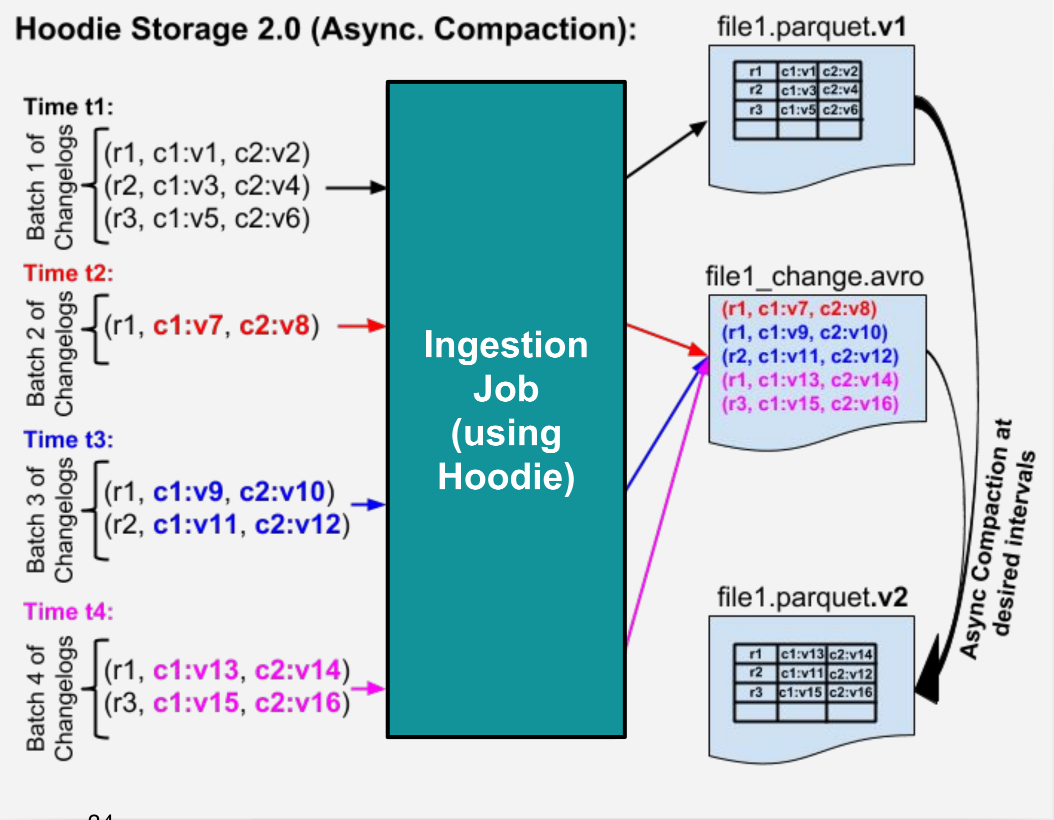
- Copy-on-write solution
- Rewriting Parquet files on updates/deletes
    - 1GB file very expensive
- Output Partition + Row_Key are required
    - Supports per partition index
    - Can we get rid of output partition?



What's coming next - Generation 4 (Ongoing effort)

Hoodie Storage 2.0:

- Merge-on-Read solution
- Have row-based delta file + Parquet file
    - Merge only when the cost of rewrite is amortized
- Merge on Query side
    - Provides 5-10min hadoop data
- Add Global Index

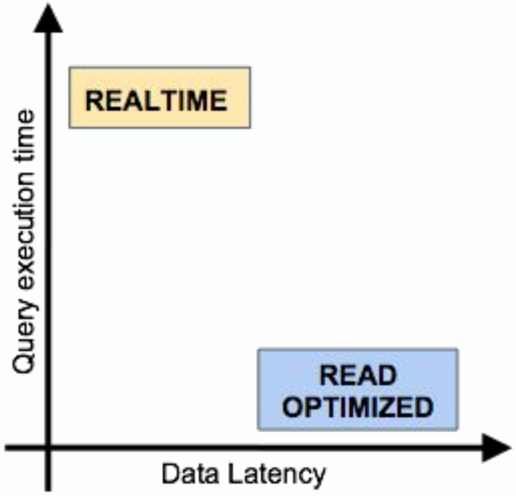


What’s coming next - Generation 4 (Ongoing effort)

Be flexible with users:

- Hudi’s supported different Storage Types and Views

Storage Type	Supported Views
Storage 1.0 (Copy On Write)	Read Optimized, ChangeLog View
Storage 2.0 (Merge On Read)	Read Optimized, RealTime, ChangeLog View



UBER

Data @ Uber: Lessons learned

Data @ Uber - Lessons Learned

1. **Investigating your data/use cases and <u>finding the required primitives</u> pays back huge**
   - With <u>GDPR requirement</u>, Having Update/Delete on the entire Hadoop dataset is life-saving
2. **<u>Data Quality</u> will be an ongoing effort**
   - <u>Enforce schema</u> (mandatory and pre-defined) as early as possible
   - Move beyond type checking and into <u>semantic checking</u> (define your own data types)
   - This is the key distinction between <u>garbage data</u> and a <u>real data-driven company</u>
3. **<u>Standardize</u> everything as soon as possible**
   - Don't make <u>exceptions</u> (it always comes back at you)
   - This is the key to having reliable Big data that can scale while being efficient
   - This is the key to have happy data users and to be able to educate them on how to use your data

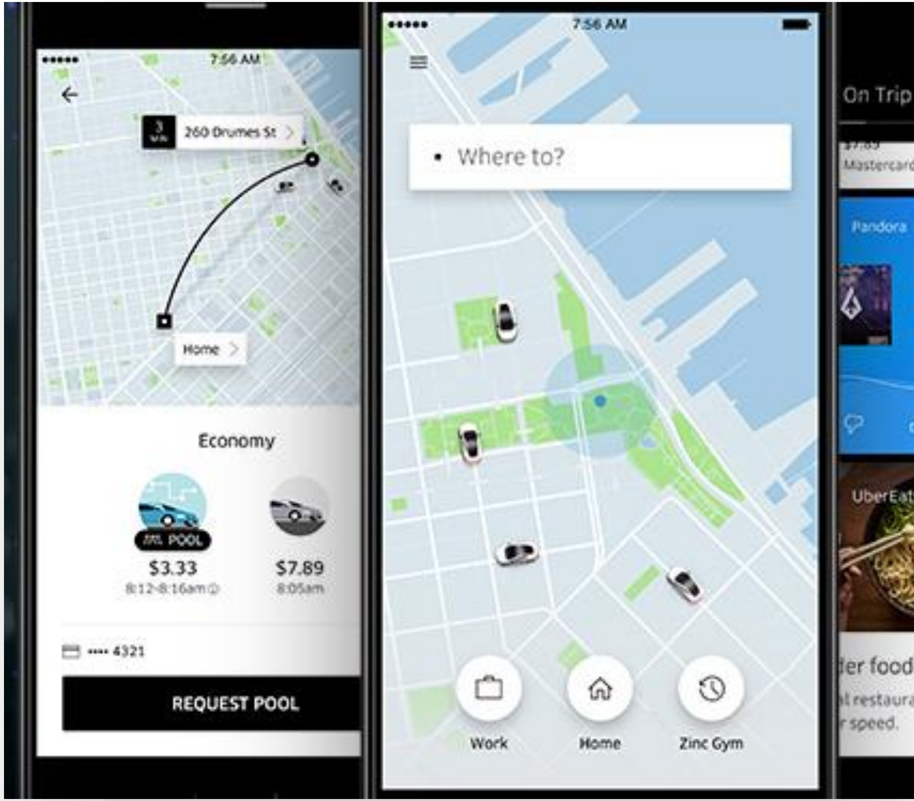
Data @ Uber - Lessons Learned

4. **Ensure you have a solid data <u>retention policy</u> as well as a standard <u>data model</u> as early as possible**
    - Retention from beginning saves you $ on wasted space and educates users to not waste
5. **Track all related data <u>metadata</u>**
    - Who owns what data, data lineage, data content, data access,...
6. **Invest in a good <u>data pipeline monitoring</u>**
    - Define your terminology and stick to it (<u>Freshness</u>, <u>Latency</u>, <u>Completeness</u>, Late-arriving-data,...)
    - Detects many corner cases and lets you solve the issue before it affects your users
7. **Minimize your dependency on <u>user-defined values</u>**
    - User-defined values always break your job
    - Replace them by <u>system-defined values</u> as much as possible (e.g. user define ts vs system ts)
8. **Pay attention to <u>notion of time in your data</u> and educate users on those**

Hadoop Platform @ Uber

Want to be part of Gen.4 or beyond?

- **Come talk to me**
  - Office Hours: 11:30am - 12:10 pm
- **Positions in both SF & Palo Alto**
  - email me: reza@uber.com



Uber's Data Journey:
100+ PB with Minute Latency

UBER

reza@uber.com

Further references

1. Open-Source Hudi Project on Github
2. "Hoodie: Uber Engineering's Incremental Processing Framework on Hadoop", Prasanna Rajaperumal, Vinoth Chandar, Uber Eng blog, 2017
3. "Uber, your Hadoop has arrived: Powering Intelligence for Uber's Real-time marketplace", Vinoth Chandar, Strata + Hadoop, 2016.
4. "Case For Incremental Processing on Hadoop", Vinoth Chandar, O'Reily article, 2016
5. "Hoodie: Incremental processing on Hadoop at Uber", Vinoth Chandar, Prasanna Rajaperumal, Strata + Hadoop World, 2017.
6. "Hoodie: An Open Source Incremental Processing Framework From Uber", Vinoth Chandar, DataEngConf, 2017.
7. "Incremental Processing on Large Analytical Datasets", Prasanna Rajaperumal, Spark Summit, 2017.
8. "Scaling Uber's Hadoop Distributed File System for Growth", Ang Zhang, Wei Yan, Uber Eng blog, 2018

Further references

9. “Hadoop Infrastructure @Uber Past, Present and Future”, Mayank Bansal, Apache Big Data Europe , 2016.
10. “Even Faster: When Presto Meets Parquet @ Uber”, Zhenxiao Luo, Apache: Big Data North America, 2017.
11.

Extra slides

Data @ Uber: Generation 2 (2015-1016)

But soon, a new set of Pain Points showed up:

Gen. 2- Pain Point #1: Reliability of the ingestion

- Bulk Snapshot based data ingestion stressed source systems
- Spiky source data (e.g. Kafka) resulted in data being deleted before it can be written out
- Source were read in streaming fashion but Parquet was written in semi-batch mode

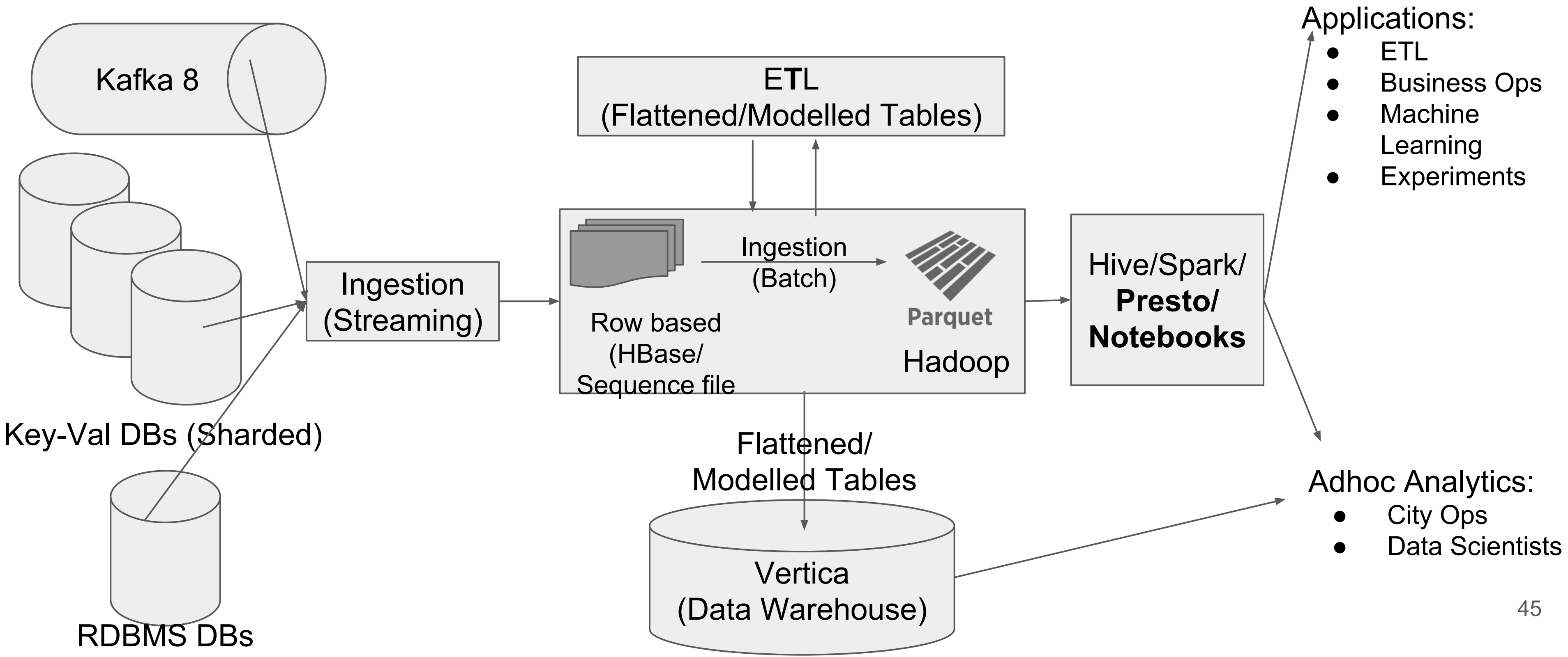
Gen. 2- Pain Point #2: Scalability

- Small file issue of HDFS started to show up (requiring larger Parquet files)
- Ingestion was not easily-scalable due to:
  - involving streaming AND/OR batch modes
  - Running mostly on dedicated HW (Needed to set it up in new DCs without YARN)
  - Large sharded Key/Val provided changelogs that needed to be merged/compacted

Gen. 3- Pain Point #3: Queries too slow

- Single choice of query engine

Data @ Uber: Generation 2.5 (2015-1016)



Data @ Uber: Generation 2.5 (2015-1016)

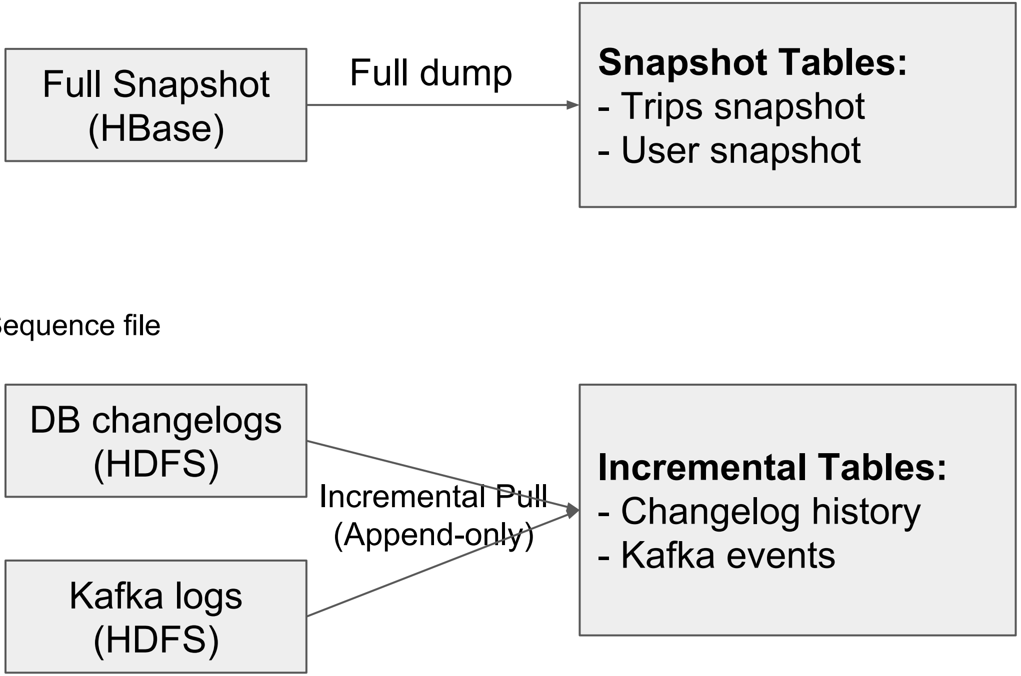
Main Highlights

- Presto added as interactive query engine
- Spark notebooks added to encourage data scientists to use Hadoop
- Simplified architecture: 2-Leg Data Ingestion
    - Get raw data into Hadoop, then do most of work as batch jobs
- Gave us time to stabilize the infrastructure (Kafka,....) & think long-term
- Reliable data ingestion with no data loss
    - since data was streamed into Hadoop with minimum work

Data @ Uber: Generation 2.5 (2015-1016)

2-Leg data ingestion:

- Leg1:
  - Running as streaming job on dedicated hardware
  - No extra pressure on the source (especially for Backfills/Catch-up)
  - Fast streaming into row-oriented storage - HBase/Sequence file
  - Can run on DCs without YARN etc
- Leg 2:
  - Running as batch jobs in Hadoop
  - Efficient especially for Parquet writing
  - Control Data Quality -
    - Schema Enforcement -
    - Cleaning JSON -
    - Hive Partitioning
  - File Stitching -
    - Keeps NN happy & queries performant



Data @ Uber: Generation 2.5 (2015-1016)

Hive:

- Powerful, scales reliably
- But slow

Vertica:

- Fast
- Can’t cheaply scale to x PB

Spark Notebooks

- Great for Data Scientists to prototype/explore data

Presto:

- Interactive queries (fast)
- Deployed at scale and good integration with HDFS/Hive
- Doesn’t require flattening unlike Vertica
- Supported ANSI SQL
- Have to improve by adding:
  - Support for geo data
  - Better support for nested data types

Data @ Uber: Generation 2.5 (2015-1016)

Solved issues from Generation 2:

Gen. 2- Pain Point #1: ~~Reliability of the ingestion~~ -> solved

- ~~Bulk Snapshot based data ingestion stressed source systems~~
- ~~Spiky source data (e.g. Kafka) resulted in data being deleted before it can be written out~~
- ~~Source were read in streaming fashion but Parquet was written in semi-batch mode~~

Gen. 2- Pain Point #2: ~~Scalability~~ -> solved

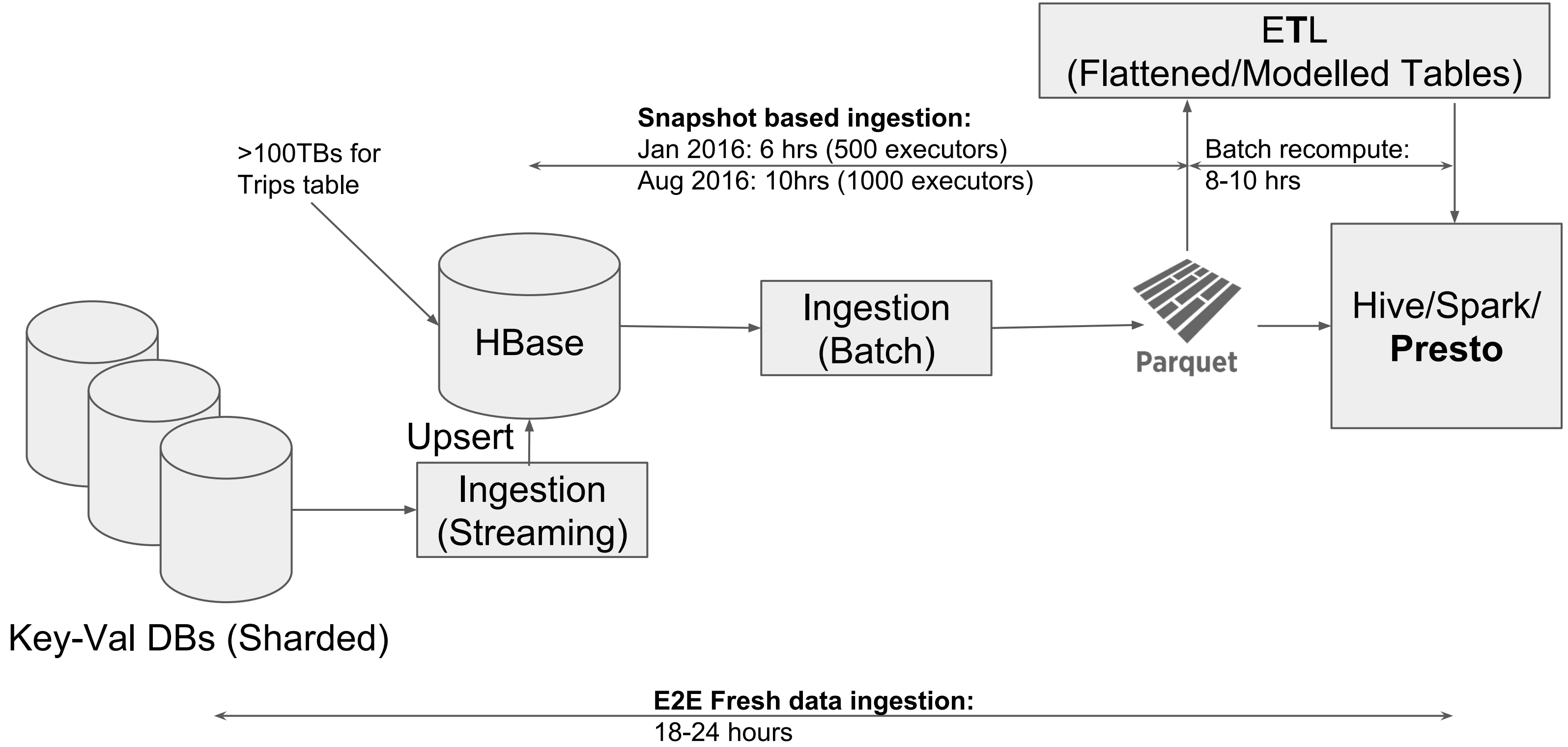
- ~~Small file issue of HDFS started to show up (requiring larger Parquet files)~~
- Ingestion was not easily-scalable due to:
  - ~~involving streaming AND/OR batch modes~~
  - ~~Running mostly on dedicated HW (Needed to set it up in new DCs without YARN)~~
  - ~~Large sharded Key/Val provided changelogs that needed to be merged/compacted~~

Gen. 2- Pain Point #3: ~~Queries too slow~~ -> solved

- ~~Limited choice of query engine~~

Data @ Uber: Generation 2.5 (2015-1016)

Pain points of snapshot-based DB ingestion:



Data @ Uber: Generation 2.5 (2015-1016)

But soon, a new set of Pain Points showed up:

Gen. 2.5- Pain Point #1: Scalability

- HDFS IO pressure since raw data was stored twice (both in row format and Parquet)
- Data ingestion pipelines became very source-specific with increased maintenance cost

Gen. 2.5- Pain Point #2: Data Latency too high

- snapshot based ingestion results in delayed fresh data (12-24hrs to get a new snapshot)
  - Even for append-only part, extra hop adds latency
  - Required async stitcher to avoid small file issue

Gen. 2.5- Pain Point #3: Updates became a big problem

- Updates are natural part of our data

Gen. 2.5- Pain Point #4: Late-arriving data also very common

- Late-arriving data because of late production time or data getting stuck in the pipeline

Gen. 2.5- Pain Point #5: ETL/ Modelling became the bottleneck

- Since most of ETL/Modelling was snapshot based (running daily off raw tables)
- Need for incremental computation to update modeled tables at hourly rate

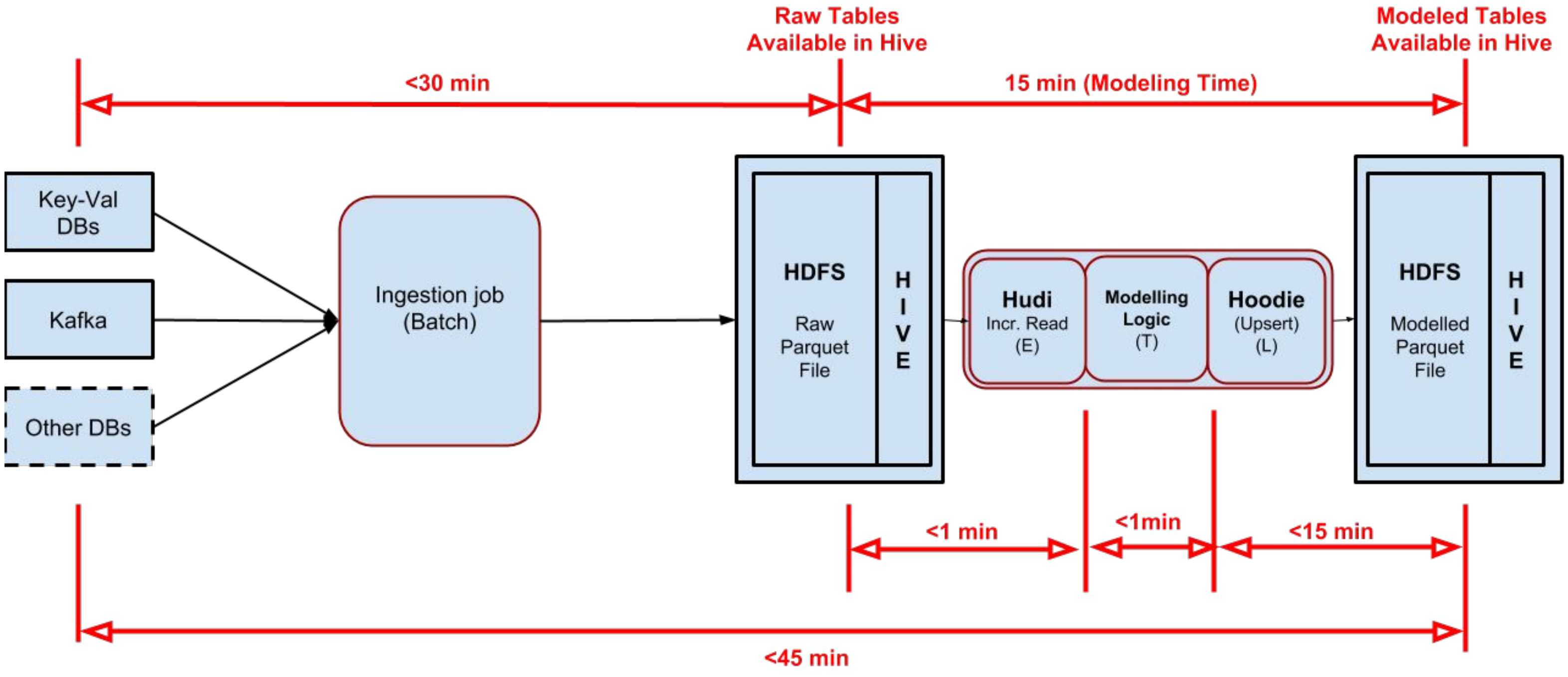
Let's rebuild for long term - Generation 3 (2017-present)

Any work-around for snapshot-based ingestion?

1. **Directly Query HBase**
    - Range scan will make it a bad fit
    - Lack of support for nested data
    - Significant operational overhead for 100 PB
2. **Don't support Snapshot view and only provide logs**
    - Users need the merged view and will have to do it in their queries which makes it inefficient
    - Merging can be done inconsistency resulting in data correctness
3. **Use specialized analytical DBs**
    - Can't bypass HDFS since we still need to join with other data in HDFS
    - Not all data fits into memory and many queries will fail
    - Leads to lambda architecture issue and multiple copies of the same data

Data @ Uber: Generation 3 (2017-present)

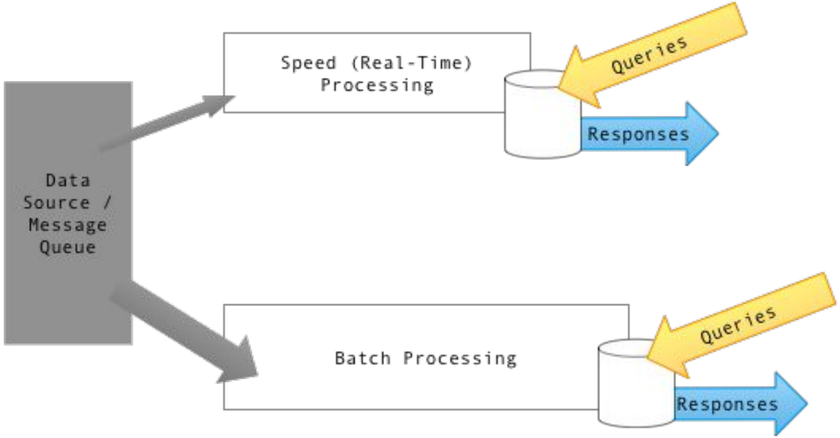
How does Incremental Ingestion in Gen 3 change data freshness/Latency?

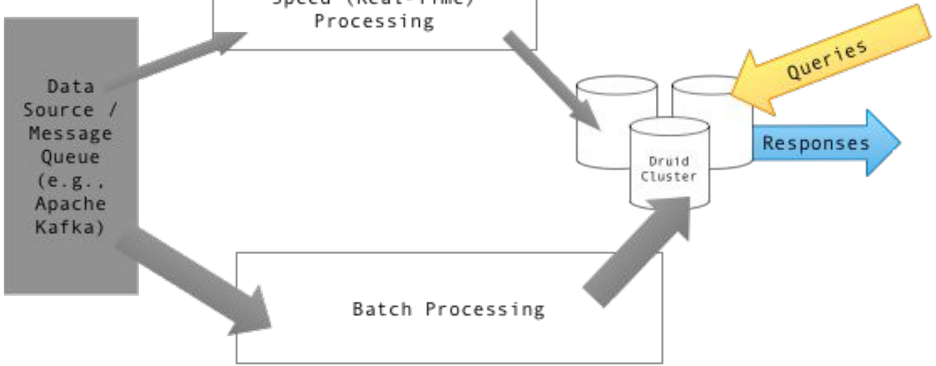


Data @ Uber: Generation 3

What does Incremental Processing mean:

Lambda architecture:





Data @ Uber: Generation 3

Stream/Batch processing Trade off:

- Latency
- Completeness
- Cost (Throughput/efficiency)

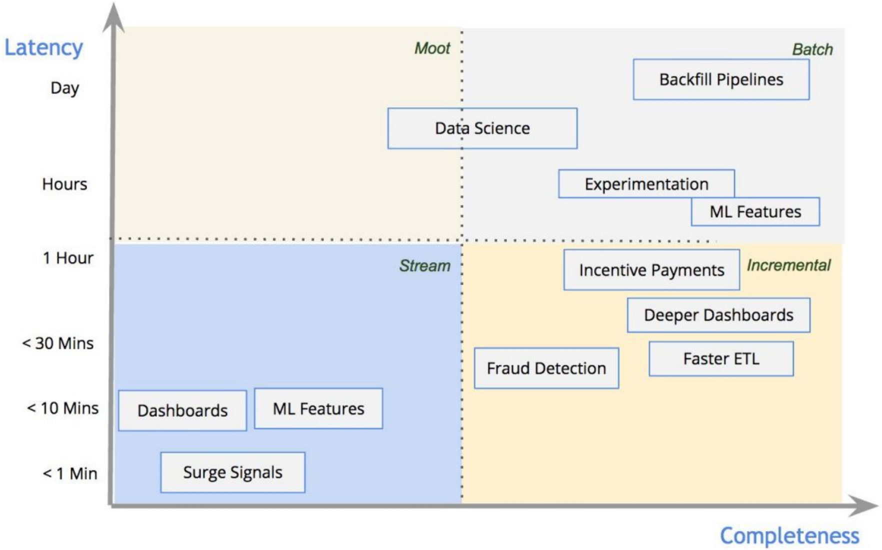
Operation challenges in Streaming & Batch:

- Projections (Streaming:Easy   Batch:Easy)
- Filtering (Streaming:Easy   Batch:Easy)
- Aggregations (Streaming:Tricky   Batch:Easy)
- Window (Streaming:Tricky   Batch:Easy)
- Joins (Streaming:HARD   Batch:Easy)

Data @ Uber: Generation 3

Do we need Streaming, Batch or Incremental?

- Need to investigate your use cases (based on latency vs Completeness)

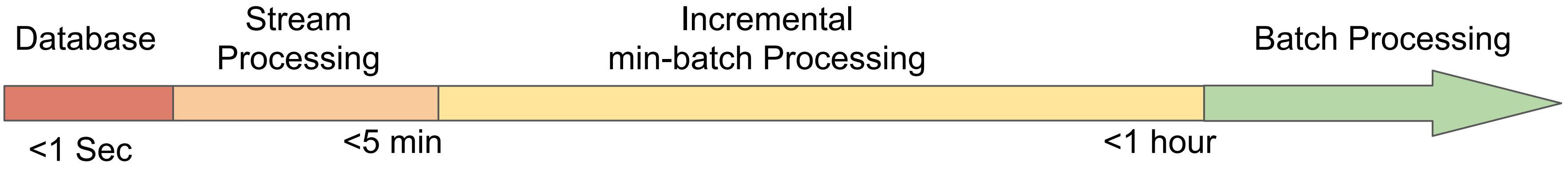


- Very distinct uses cases for Streaming
- Very distinct use cases for Batch
- A lot of use cases that can benefit from incremental mode

Data @ Uber: Generation 3: Provide Incremental processing

What exactly is Incremental mode?

- Mini-batch jobs that pulls out only changed data
- Provides high completeness (compared to streaming mode)
- Supports all hard operations as any other batch job (like multi-table joins,....)



Data @ Uber: Generation 3: Provide Incremental processing

How does Incremental mode help efficiency?

- Read only what you need by using Columnar file formats
- Simple solution for all types of queries (joins, …)
- Consolidation of Compute & Storage for all use case (exploratory, interactive,....)