Hudi

Large-scale, Near real-time pipelines at Uber

Nishith Agarwal, Vinoth Chandar

Session hashtag: #SAISEco10

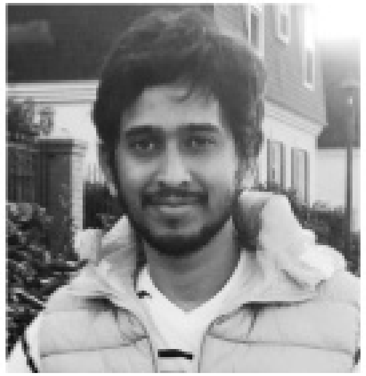
UBER

Speaker Intros

Who are we?

Nishith Agarwal

Snr Engineer on the Hadoop Platform team, working on Hudi & hadoop ingestion at large.



Vinoth Chandar

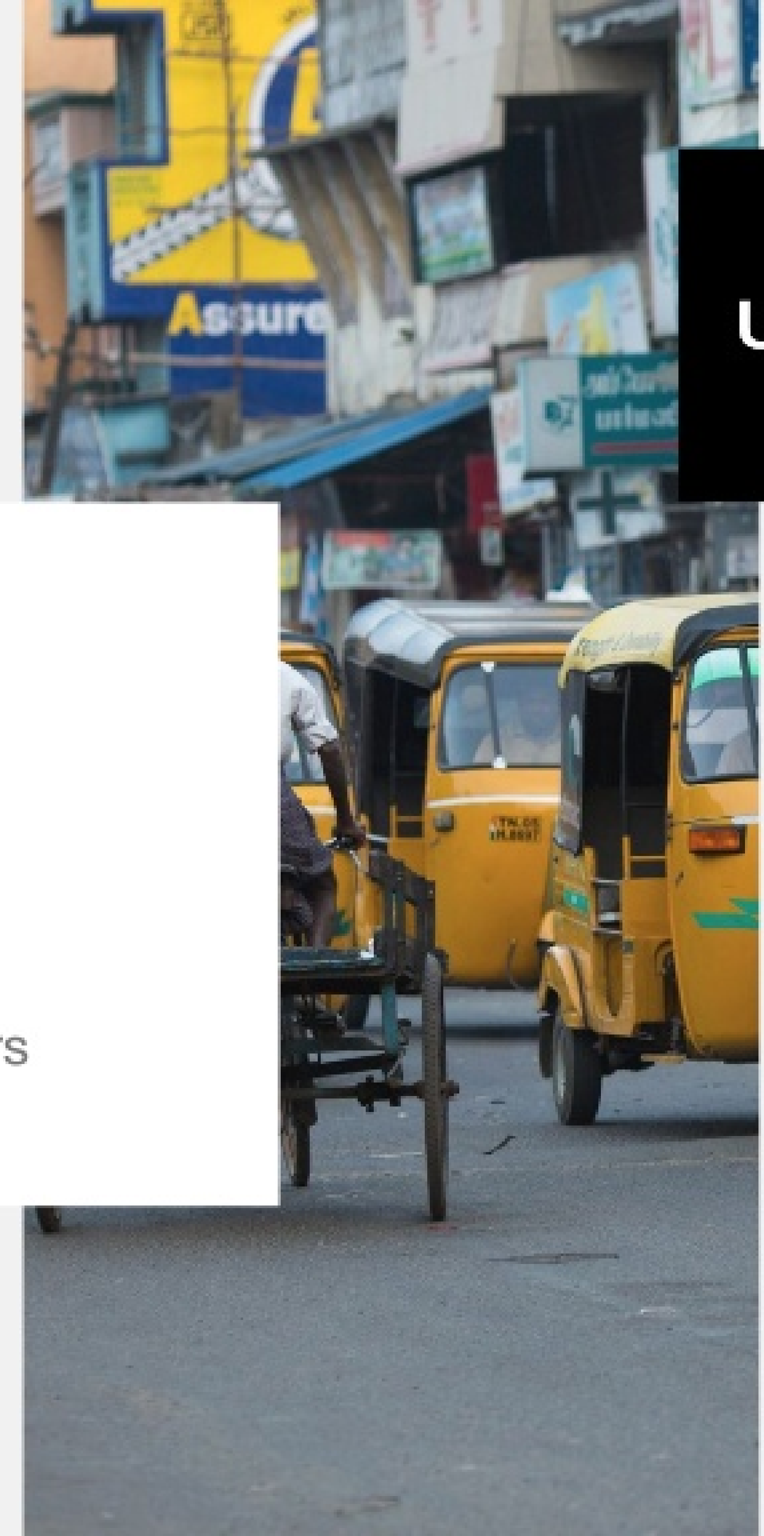
Staff engineer on Infrastructure org, working on two core technologies : Hudi/BigData storage, Network protocols for the edge

Uber’s scale

Smarter cities of the future

On a snowy Paris evening in 2008, Travis Kalanick and Garrett Camp had trouble hailing a cab. So they came up with a simple idea—press a button, get a ride.

700+	70+	2M+
Cities	Countries	Driver partners



UBER

Agenda

- What's Incremental Processing?
- Hudi - Overview
- Recipe : Database Ingestion
- Recipe : Deduping Logs
- Recipe : Incremental ETL
- OSS Community

UBER

Spark Pipelines

Different Shapes & Sizes

Read, Transform, Write

- Row level (Projections/Filters)
- Aggregations (group-by, sums, averages)
- Windowing/Joins

Batch Model

- Read : All data for day/hour(s)
- Transform : Recompute all results
- Write : Replace entire day/hour(s) results

Streaming Model

- Read : New data since last compute
- Transform : Compute result deltas using state store
- Write : Update new results in sink

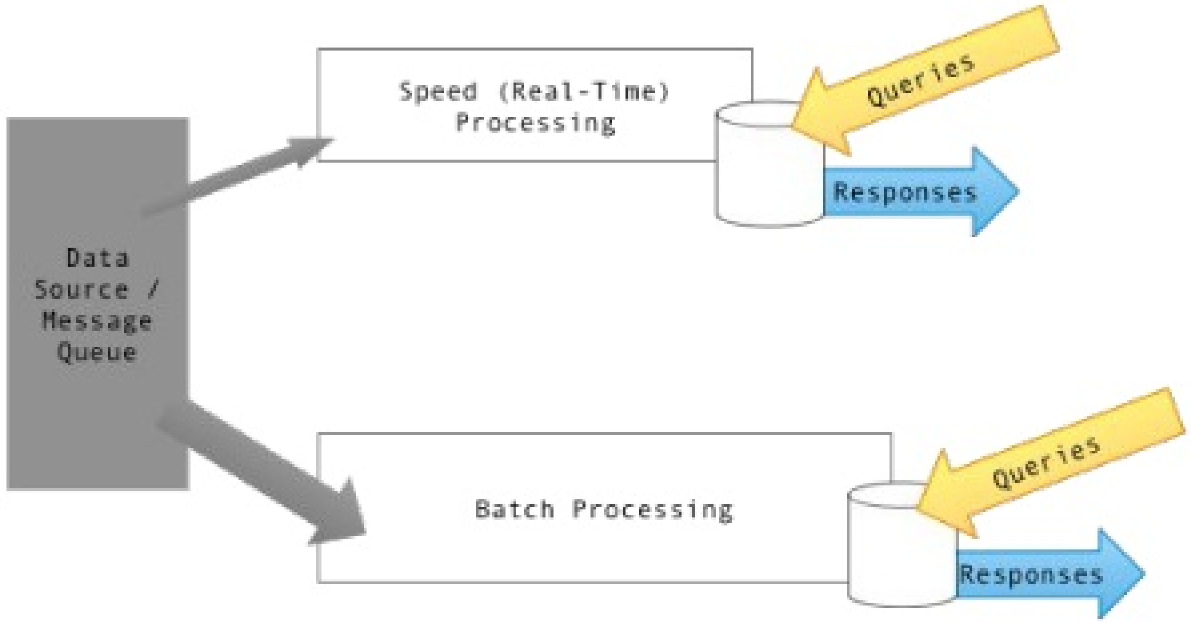


Fig: Typical lambda architecture employing both pipelines

Spark Pipelines

Key Levers

Latency

How fresh are the computed results?

Completeness

How accurate are the computed results?

Scale/Cost

For given latency/completeness, how much data can be read/written at reasonable cost?

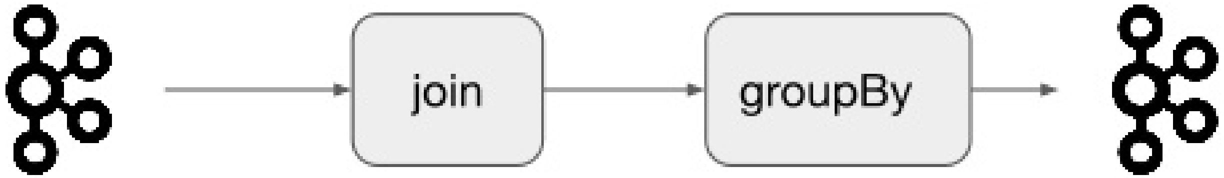


Fig: Multi way join/aggregation, at low latency, high cost and moderate/low completeness

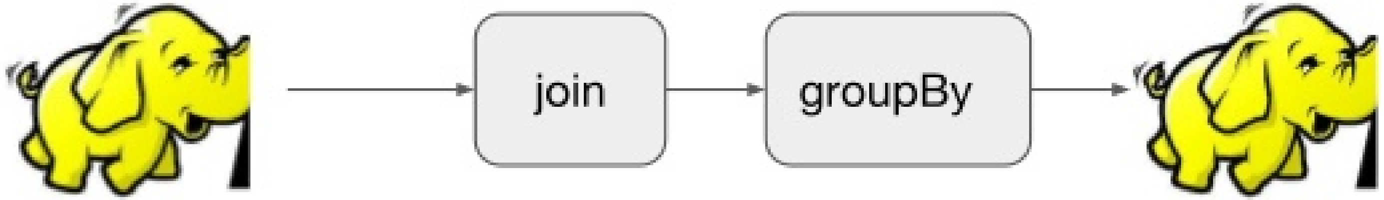


Fig: Multi way join/aggregation, at high latency, low cost and high completeness

How about large data@low latency ?

Terabytes to Petabytes

Streaming Model

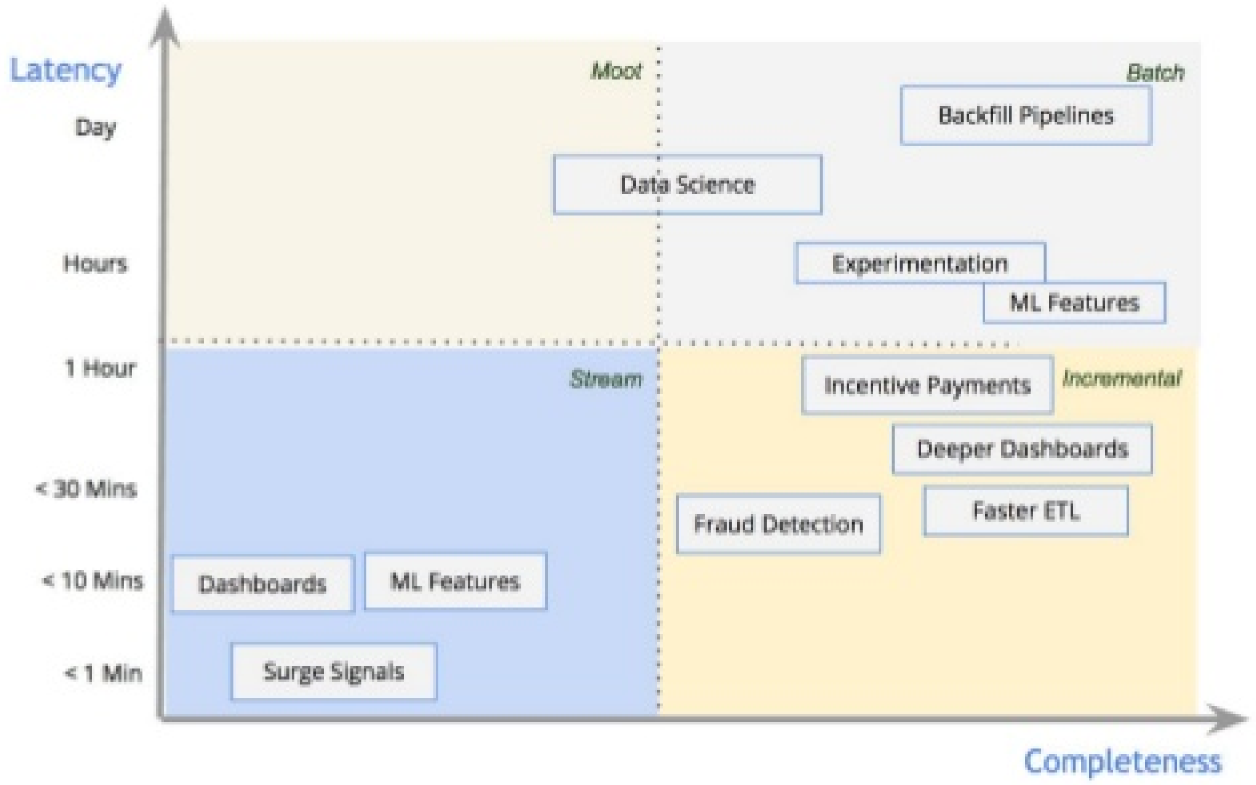
- Great latency
- Smaller datasets (TBs)

Batch Model

- Great efficiency
- Large datasets

It’s needn't be a dichotomy!

- Some don’t fit in either!
- Support entire spectrum of use-cases



Incremental Model

Streaming Style Processing On Batch Data

Near real-time results

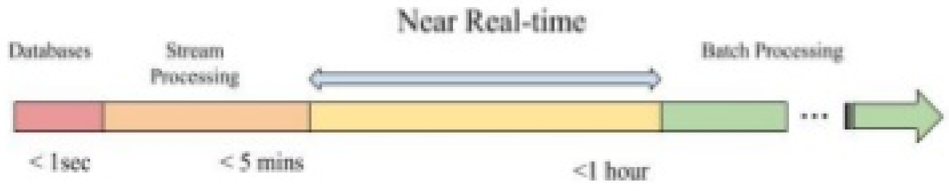
- Mini batch jobs, every few minutes

Upsert (Primitive #1)

- Modify processed results
- Like state stores in stream processing

Incremental Pull (Primitive #2)

- Log stream of changes, avoids costly scans
- Faster flow of data to next stage in dataflow



Incremental Model

Trade-offs

Latency

Streaming < Incremental < Batch

Completeness

Batch* < Incremental < Streaming

Scale

Streaming < Incremental < Batch

Cost

Batch < Incremental < Streaming

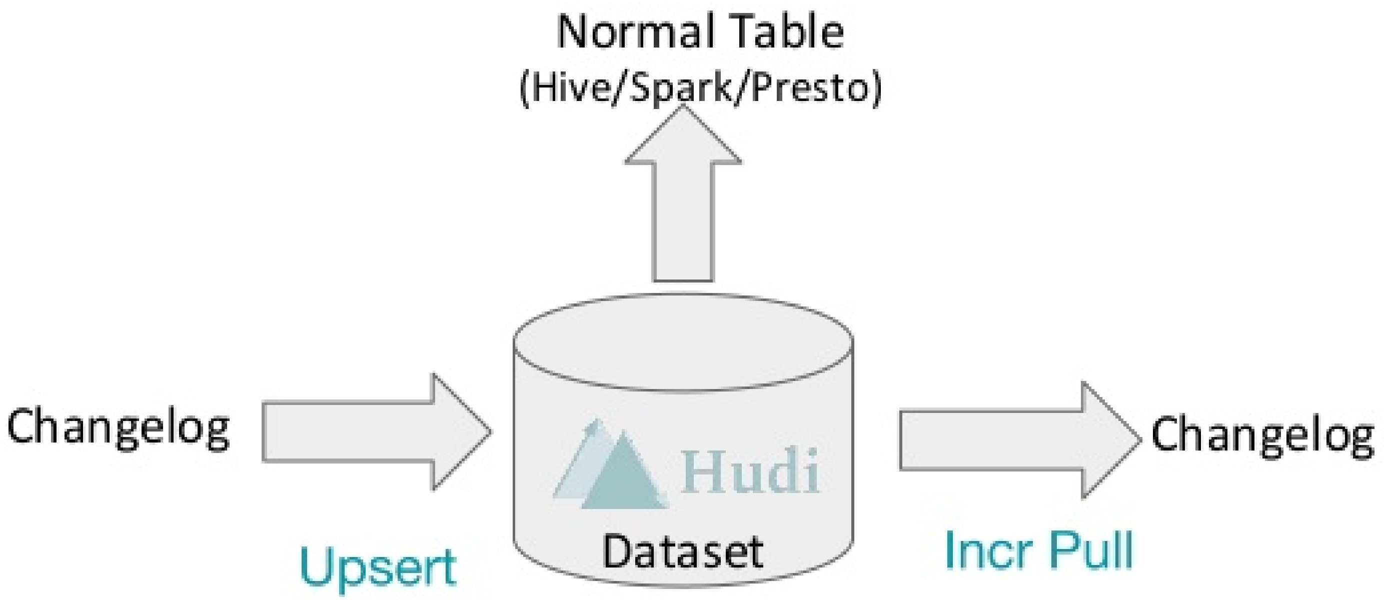
HUDI : Hadoop Upserts anD Incrementals

Turn batch jobs to incremental model

- Improve latency by incorporating only deltas
- Scale more by avoiding recomputation, lowering cost

Spark Library for

- Mutations to datasets
- Changelogs from datasets
- Manage files efficiently
- Provide snapshot isolation



Open Source

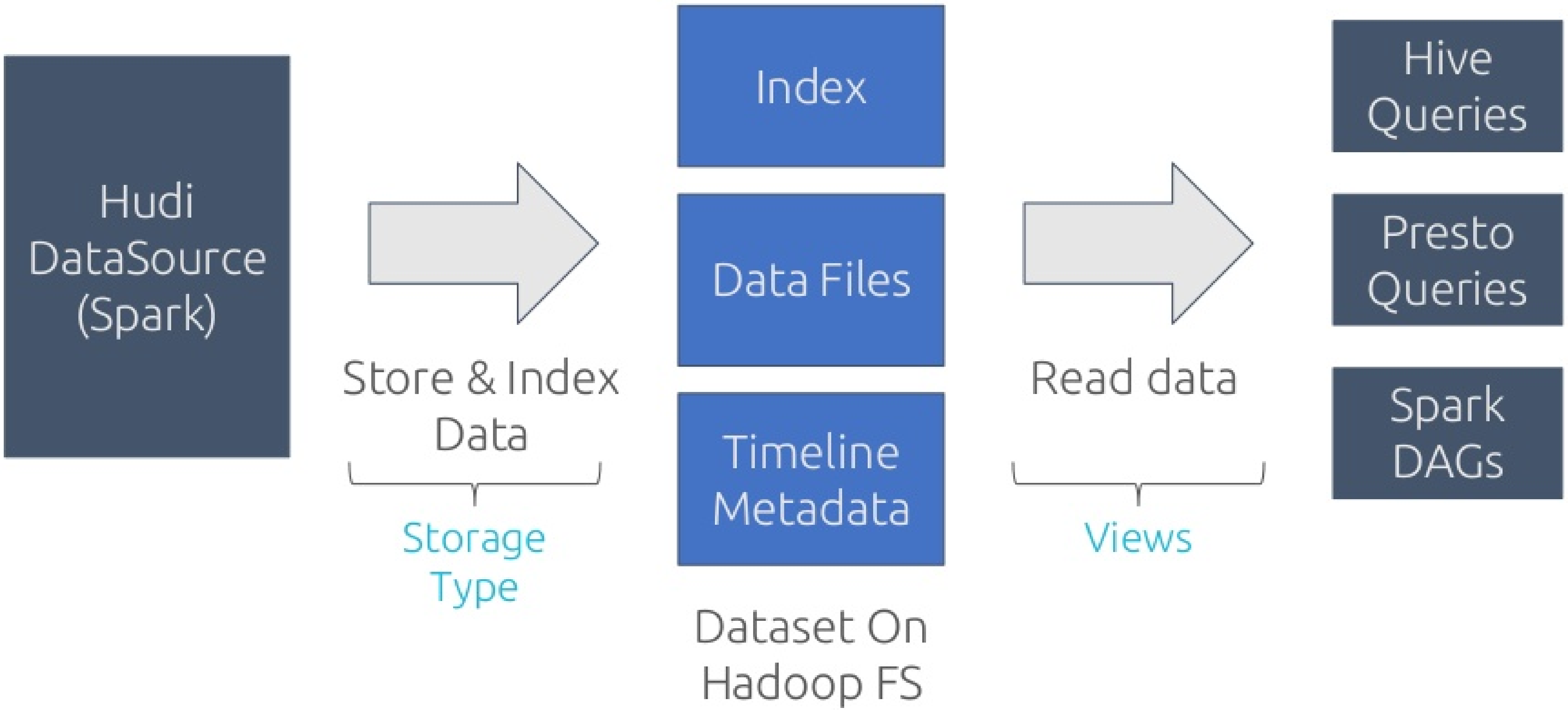
- https://github.com/uber/hudi
- https://eng.uber.com/Hudi

HUDI

@Uber

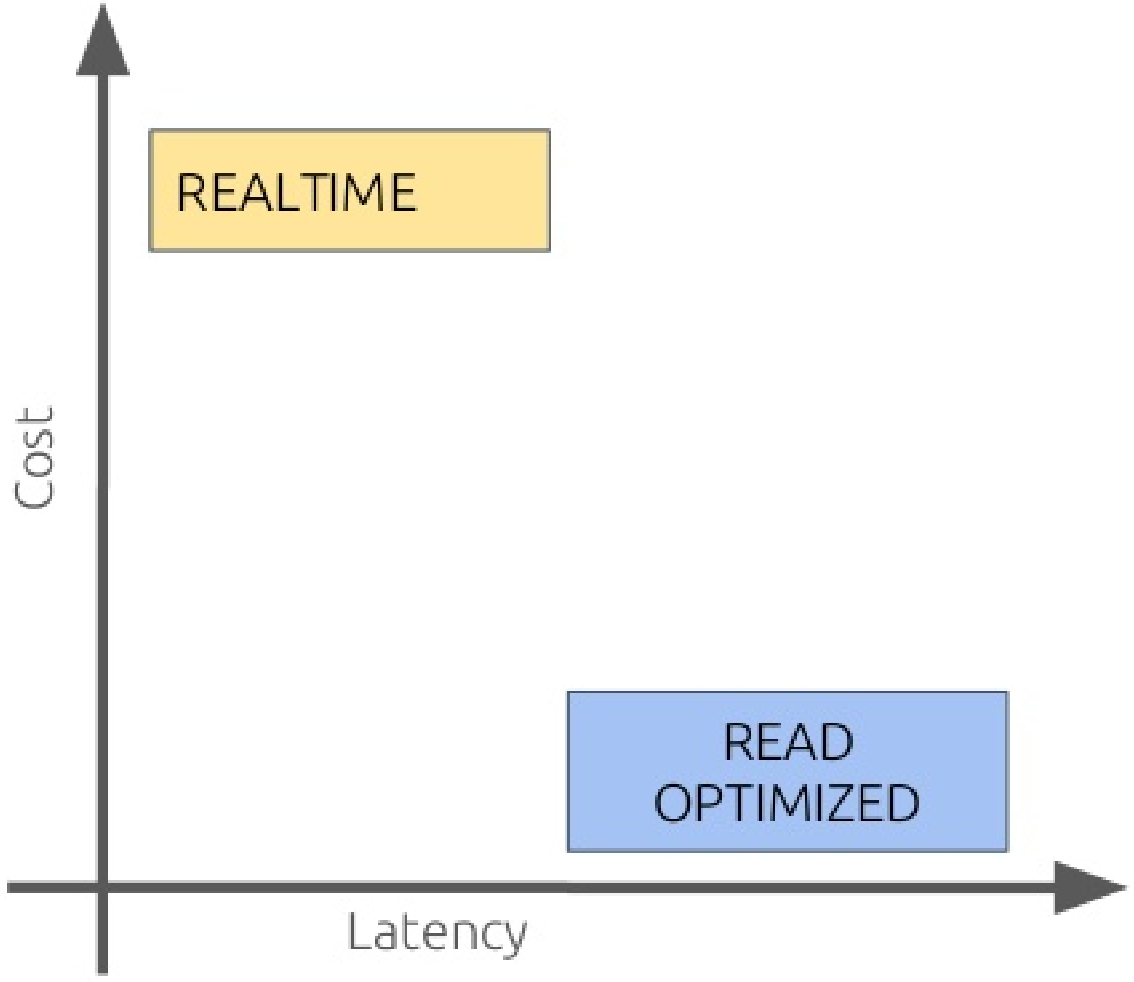
10s PB	1000s	100s TB
Total Storage	Ingest Pipelines	Ingested/Day
100s	10000+	4-30 mins
ETL Pipelines	Yarn Containers	Latency Range

HUDI
Components



Hudi Views

Different options to view the data



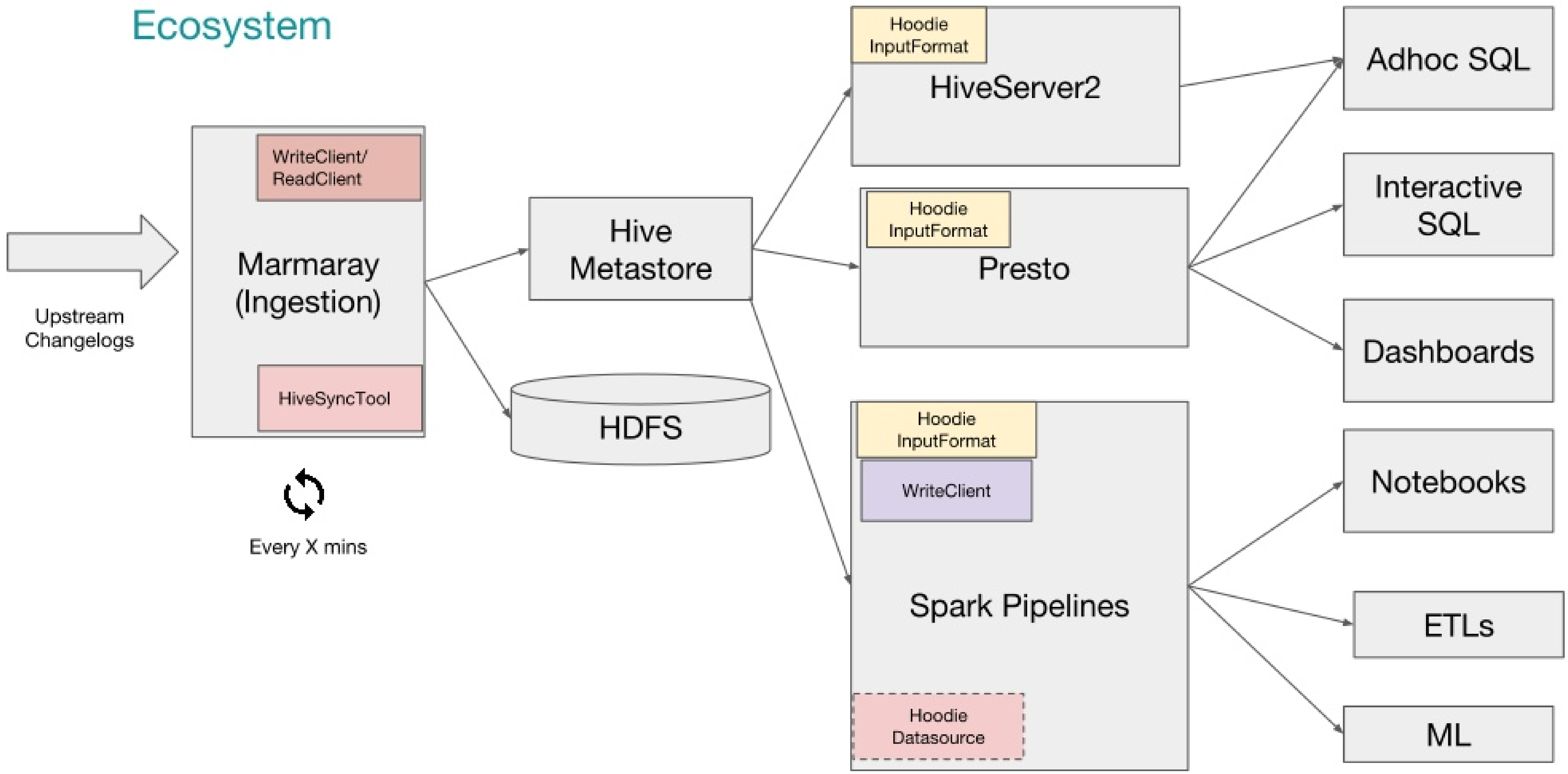
3 Logical views Of Dataset

Read Optimized View
- Raw Parquet Query Performance
- Targets existing Hive tables

Real Time View
- Hybrid of row & columnar data
- Brings near-real time tables

Incremental View
- Stream of changes to a dataset
- Enables Incremental Pull

HUDI
Ecosystem



HUDI

Tools

HiveSyncTool

- Register Hive tables to access Hudi views
- Registers a RO & RT table by default
- Handles schema evolution

DeltaStreamer

- Ingest tool supporting DFS & Kafka sources
- Json and Avro data formats supported

Spark Datasource

- Custom incremental pipelines
- Jobs needing upserts into sink

CLI

- CLI for inspecting timeline
- Repair tools
- 1-time data imports

HiveIncrementalPuller

- Incrementally consume data from Hive via HQL

SnapshotCopier

- Consistently Copy/Backup Hudi datasets

Recipe: Database Ingestion

Problem

OLTP Databases

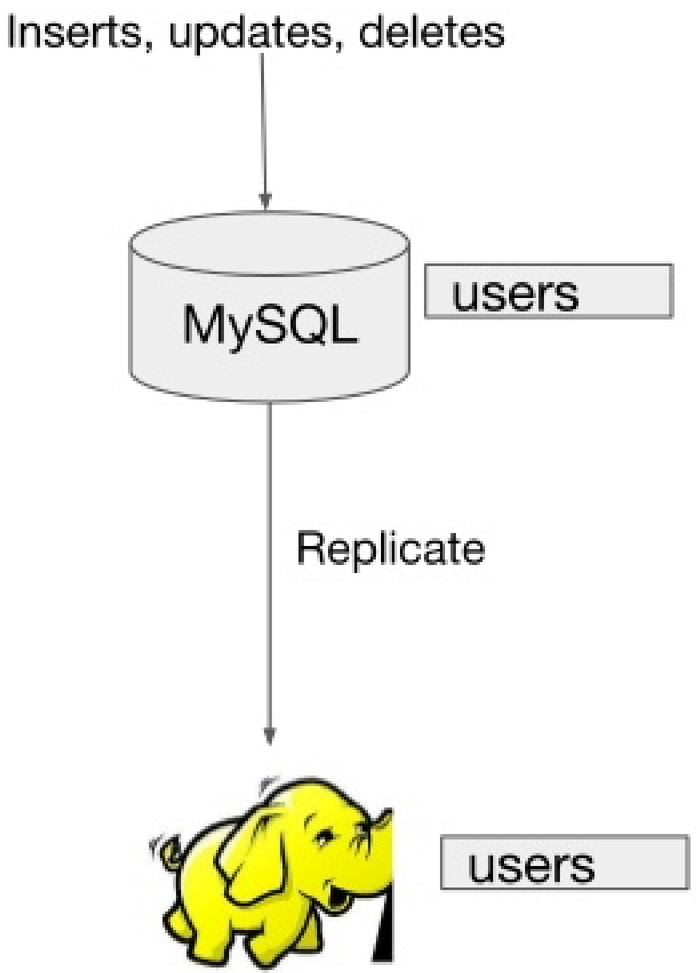
- MySQL
- NoSQL

High-value data

- User information
- Trip, transaction logs
- Heavily joined

Replicate CRUD operations

- Hadoop/Hive tables need to be kept upto date



userID	int
country	string
last_mod	long
...	...

Recipe: Database Ingestion

Typical Approach

Bulk Load

- Every few hours, copy all data
- Inefficient, expensive

Incremental Extract + Bulk Merge

- Tail table or redo logs to obtain changes C
- Merge C into existing table T
- Needs full table scan of T
- Typically done every few hours!

What If we want fresher data?

- Do analytics on a read-only copy!
- But, scales poorly for analytical scans!

Recipe: Database Ingestion

Sqoop + Datasource

```
// Command to extract incrementals using sqoop
bin/sqoop import \
  -Dmapreduce.job.user.classpath.first=true \
  --connect jdbc:mysql://localhost/users \
  --username root \
  --password ******* \
  --table users \
  --as-avrodatafile \
  --target-dir \
s3:///tmp/sqoop/import-1/users
```

Extract new changes to users table in MySQL, as avro data files on DFS

```
// Spark Datasource
Import com.uber.hoodie.DataSourceWriteOptions._
// Use Spark datasource to read avro
Dataset<Row> inputDataset
spark.read.avro('s3://tmp/sqoop/import-1/users/*');

// save it as a Hoodie dataset
inputDataset.write.format("com.uber.hoodie")
  .option(HoodieWriteConfig.TABLE_NAME, "hoodie.users")
  .option(RECORDKEY_FIELD_OPT_KEY(), "userID")
  .option(PARTITIONPATH_FIELD_OPT_KEY(),"country")
  .option(PRECOMBINE_FIELD_OPT_KEY(), "last_mod")
  .option(OPERATION_OPT_KEY(), UPSERT_OPERATION_OPT_VAL())
  .mode(SaveMode.Append);
  .save("/path/on/dfs")
```

Use your fav datasource to read extracted data and directly "upsert" the users table on DFS/Hive

Completes in few mins!!

Recipe: Database Ingestion

Sqoop + DeltaStreamer

```
// Deltastreamer command to ingest sqoop incrementals
spark-submit \
  --class com.uber.hoodie.utilities.deltastreamer.HoodieDeltaStreamer \
  /path/to/hoodie-utilities-*-SNAPSHOT.jar` \
  --props    s3://path/to/dfs-source.properties \
  --schemaprovider-class com.uber.hoodie.utilities.schema.FilebasedSchemaProvider \
  --source-class com.uber.hoodie.utilities.sources.AvroDFSSource \
  --source-ordering-field last_mod \
  --target-base-path s3:///path/on/dfs
  --target-table uber.employees \
  --op UPSERT
```

```
// dfs-source-properties
include=base.properties
# Key generator props
hoodie.datasource.write.recordkey.field=_userID
hoodie.datasource.write.partitionpath.field=country
# Schema provider props
hoodie.deltastreamer.filebased.schemaprovider.source.schema.file=s3:///path/to/users.avsc
# DFS Source
hoodie.deltastreamer.source.dfs.root=s3:///tmp/sqoop
```

Recipe: Deduping Logs

Problem

High-scale timeseries data

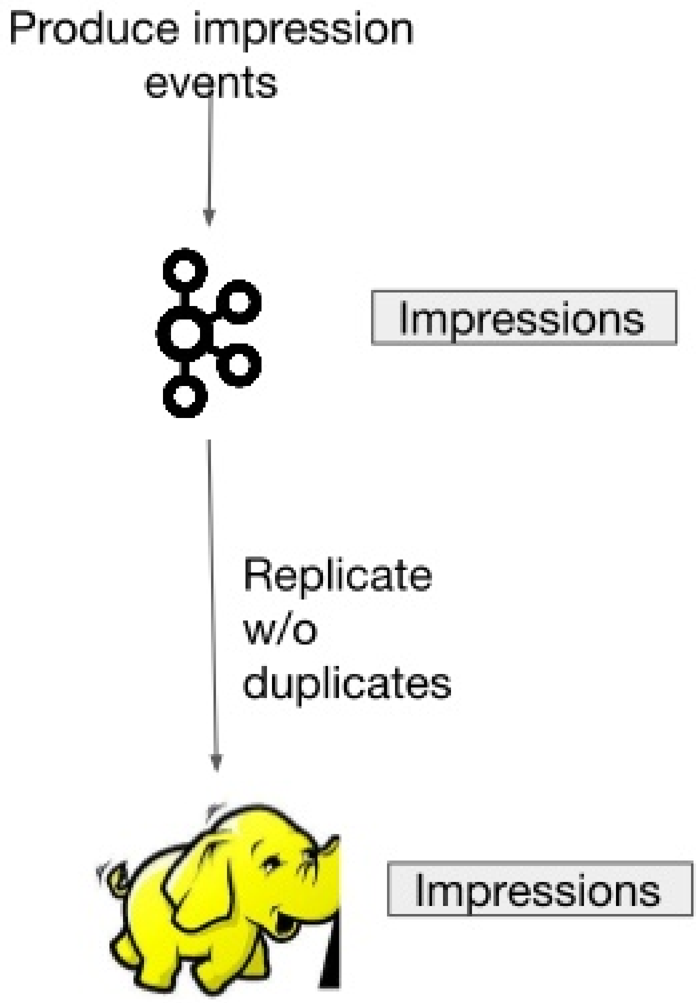
- Several billions/day
- Few millions/sec
- Heavily aggregated

Cause of duplicates

- Client retries/failures/network errors
- Atleast-once data pipes

Overcounting problems

- More impressions => more $
- Low fidelity data



id	string
datestr	string
time	long
...	...

Recipe: Deduping Logs

Typical Approach

Pass it on to clients

- Very inefficient
- Often inconsistent

Key-Value Stores

- Non trivial scaling issues
- Expensive O(num_events) storage/lookup

Periodic data cleansing

- Duplicates have leaked to consumers anyway!
- Expensive

Recipe: Deduping Logs

Filtering Duplicates

```
// Deltastreamer command to ingest kafka events, dedupe, ingest
spark-submit --class com.uber.hoodie.utilities.deltastreamer.HoodieDeltaStreamer \
  /path/to/hoodie-utilities-*-SNAPSHOT.jar` \
  --props s3://path/to/kafka-source.properties \
  --schemaprovider-class com.uber.hoodie.utilities.schema.SchemaRegistryProvider \
  --source-class com.uber.hoodie.utilities.sources.AvroKafkaSource \
  --source-ordering-field time \
  --target-base-path s3:///hoodie-deltastreamer/impressions --target-table uber.impressions \
  --op BULK_INSERT
  --filter-dupes
```

```
// kafka-source-properties
include=base.properties
# Key fields, for kafka example
hoodie.datasource.write.recordkey.field=id
hoodie.datasource.write.partitionpath.field=datestr
# schema provider configs
hoodie.deltastreamer.schemaprovider.registry.url=http://localhost:8081/subjects/impressions-value/v
ersions/latest
# Kafka Source
hoodie.deltastreamer.source.kafka.topic=impressions
#Kafka props
metadata.broker.list=localhost:9092
auto.offset.reset=smallest
schema.registry.url=http://localhost:8081
```

Recipe: Incremental ETL

Problem

Multi stage ETL DAGS

- Very common in batch analytics
- Large amount of data
- Big Spark use-case!

Derived/ETL tables

- Keep afresh with new/changed source data
- Star schema/warehousing

raw_trips

id	string
datestr	string
currency	string
fare	double

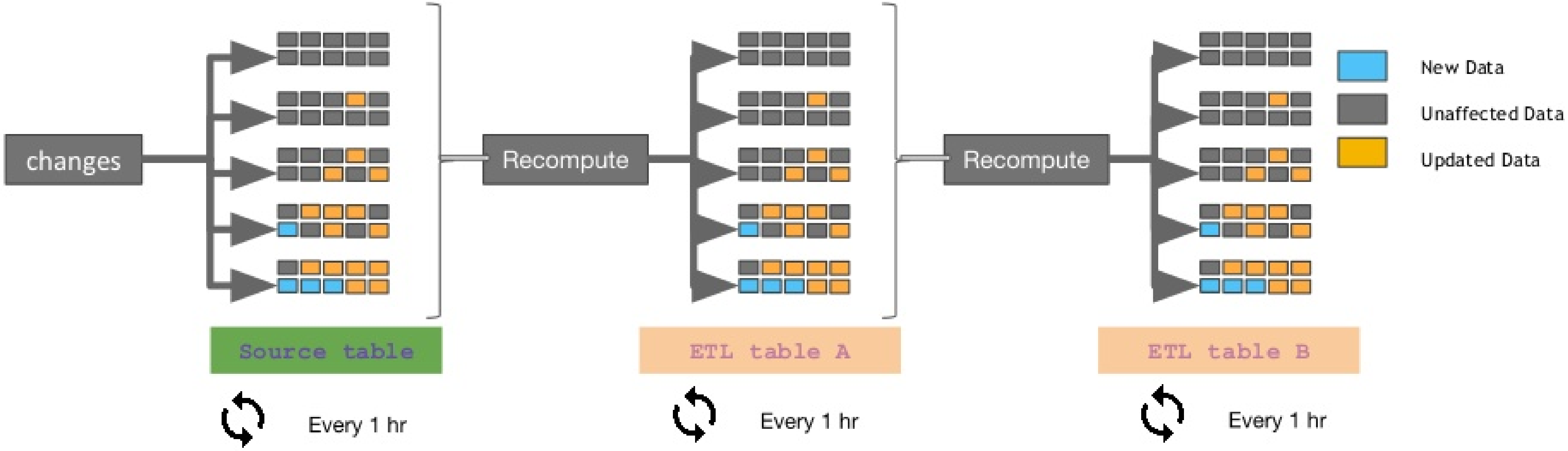
standardize_fare(row)

std_trips

id	string
datestr	string
std_fare	double
...	...

Recipe: Incremental ETL

Typical Approach



Recompute in every stage

- Heuristics => limit scan range
- Slowwww

Minimum 3 hr latency!

- Waste of resourcing in recomputing

Recipe: Incremental ETL

Incremental Spark Pipelines

Incrementally pull

- Avoid recomputes!
- Finishes in few mins!

Transform + upsert

- Avoid rewriting all data

```
// Spark Datasource
Import com.uber.hoodie.{DataSourceWriteOptions, DataSourceReadOptions}._

// Use Spark datasource to read avro
Dataset<Row> hoodieIncViewDF = spark.read().format("com.uber.hoodie")
   .option(VIEW_TYPE_OPT_KEY(), VIEW_TYPE_INCREMENTAL_OPT_VAL())
   .option(DataSourceReadOptions.BEGIN_INSTANTTIME_OPT_KEY(),commitInstantFor8AM)
   .load("s3://tables/raw_trips");

Dataset<Row> stdDF = standardize_fare(hoodieIncViewDF)

// save it as a Hoodie dataset
inputDataset.write.format("com.uber.hoodie")
  .option(HoodieWriteConfig.TABLE_NAME, "hoodie.std_trips")
  .option(RECORDKEY_FIELD_OPT_KEY(), "id")
  .option(PARTITIONPATH_FIELD_OPT_KEY(),"datestr")
  .option(PRECOMBINE_FIELD_OPT_KEY(), "time")
  .option(OPERATION_OPT_KEY(), UPSERT_OPERATION_OPT_VAL())
  .mode(SaveMode.Append);
  .save("/path/on/dfs")
```

Open Source Community

We Love Contributions!

Github

- GitHub -> https://github.com/uber/hudi
- Quickstart -> https://uber.github.io/hudi/quickstart.html
- Documentation -> https://uber.github.io/hudi/
- Slack channel -> https://hoodielib.slack.com/

Contributions

- Small & large, code & design & feature asks
- 10+ organizations incl Uber, Shopify, DoubleVerify, Vungle, Udemy..

Roadmap

- Global index to ease migrations
- Ease of managing compactions
- RTView performance

WE ARE HIRING!

Reach out to us

hadoop-platform-jobs@uber.com

Thank you. Questions?

nagarwal@uber.com
vinoth@uber.com

UBER