Hoodie

How (and Why) Uber built an Analytical datastore On Spark

Prasanna Rajaperumal, Engineer, Uber

June, 2017

What’s our problem?

Queryable State for Analytics

Analytics == Big Scans

- Super fast scans on subset of columns
- Large time ranges - Lots of data

Queryable state == mutations

- Pure Dimension Table e.g. Users
- Fact Tables that can get super large and needs a materialized view e.g. Trips
- Late Arriving Data
  - Event time vs Processing time
- Delete records (Compliance)
- Data correction upstream

source

Okay, so what did we want?

OLAP Database

- **Scale and complexity**
  - Scale horizontally [Petabytes]
  - Support Nested columns
  - Batch ingest and Analytical scans
- **Latency**
  - Ingest Latency ~ 10 minutes
  - Query Latency ~ upto 2 minutes
- **Multi tenant - High throughput**
- **Transactional - ACID**
- **Self Healing**
  - Less tunable knobs
  - Handle data skew
  - Auto scale with load
  - Failure Recovery
  - Rollback and Savepoints

source

Okay, Could you do ...?

Solutions that did not work for us

- **OLAP RDBMS**
  - Petabyte scale
  - Elastic scaling of compute
- **No/New SQL (LSM)**
  - Scan performance
  - Operations involved - Compaction
- **Hack around it**
  - Dump LSM Snapshot
  - Rewrite partitions too costly
  - Watermark - Approximations
- **Hive Transactions**
  - Hive specific solution
  - Hash bucketing - tuning?
- **Apache Kudu**
  - Separate storage server
  - Eco system support

source

Let's design what we want.
We have 20 minutes.

“Software Engineer engineers the illusion of simplicity”

Grady Booch, UML Creator

Pick the area in RUM triangle

Design choices

- **RUM Conjecture**
  - Optimize 2 at the expense of the third
- **Fast data - Write Optimized**
  - Control Read Amplification
  - Query execution cost
- **Fast Scans - Read Optimized**
  - Control Write Amplification
  - Ingestion cost
- **Choice per client/query**

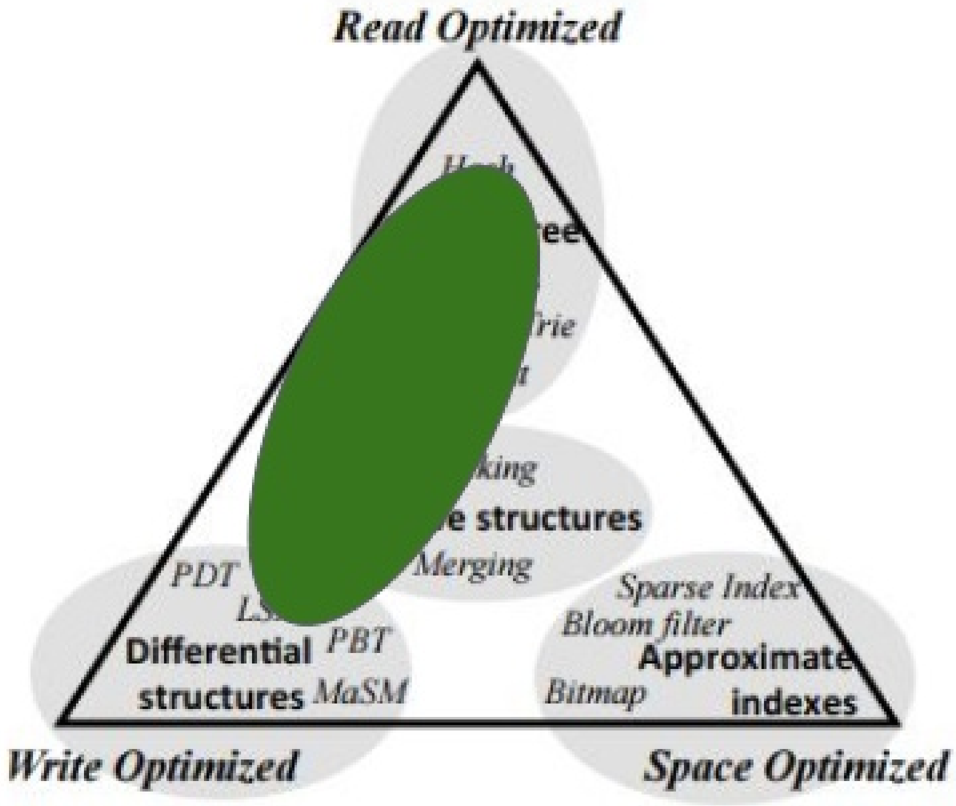


Figure 1: Popular data structures in the RUM space.

source

Pick Framework

Leveraging Spark's Elasticity + Scalability + Speed

- **Spark + DFS vs Storage Server**
  - **Batch engine vs MPP engine**
    - Throughput vs Latency
    - Flexibility to go batch or streaming
    - Dynamic Resource Allocation
  - **Complexity**
    - Static Partitioning
    - Dedicated resources
    - Consensus
  - **Scaling**
    - Auto Scaling with load using Spark
  - **Resiliency and Recovery (RDD)**
    - Simplify Application Abstraction
    - Self Healing
  - **Simplified API Layer**

Correctness - ACID

Design choices

- **Atomic ingest of a batch**
  - Based on Processing time
  - Cross row atomicity
- **Strong consistency**
- **Single Writer | Multiple Reader**
- **High query concurrency**
  - Query Isolation using Snapshot
- **Time travel**
  - Temporal queries

source

Storage

Design choices

- **Hybrid Storage**
  - Row based - Recent data
  - Column based - Cold data
- **Compactor**
- **Insert vs Update during Ingest**
  - Need for Index
- **Ingest parallelism vs Query parallelism**
  - Max file size

Partitioning

Implementation choices

- **DFS - Directory Partitioning**
  - Coarse grained
  - Need finer grained
    - Hash Bucket
    - Auto create partition on insert

Introducing **Hoodie**

Hadoop **U**psert an**D** **I**ncrementals

https://github.com/uber/hoodie

https://eng.uber.com/hoodie

How do I ingest?

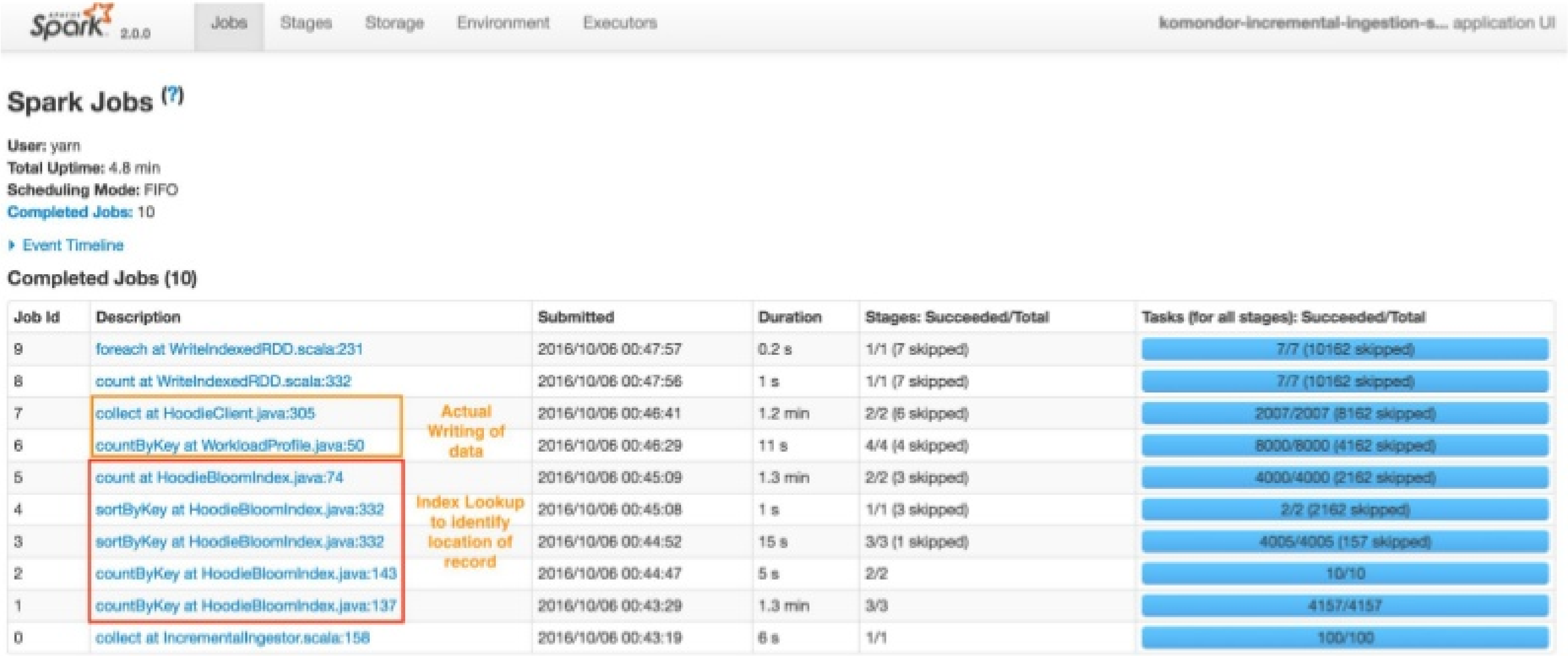
Show me the code !!!

```
HoodieWriteConfig cfg = HoodieWriteConfig.newBuilder()
             .withPath(path)
             .withSchema(schema)
             .withParallelism(500)
             .withIndexConfig(HoodieIndexConfig.newBuilder()
                   .withIndexType(HoodieIndex.IndexType.BLOOM).build())
             .withStorageConfig(HoodieStorageConfig.newBuilder()
                   .defaultStorage().build())
             .withCompactionConfig(HoodieCompactionConfig.newBuilder()
                   .withCompactionStrategy(new BoundedIOCompactionStrategy()).build())
             .build();

JavaRDD<HoodieRecord> inputRecords = … // input data
HoodieWriteClient client = new HoodieWriteClient(sc, cfg);
JavaRDD<WriteStatus> result = client. upsert(inputRecords, commitTime);
boolean toCommit = inspectResultFailures(result);
if(toCommit) {
      client.commit(commitTime, result);
} else {
      client.rollback(commitTime);
}
```

Spark DAG

How is that graph looking?



Spark Jobs (?)

User: yarn
Total Uptime: 4.8 min
Scheduling Mode: FIFO
Completed Jobs: 10

▸ Event Timeline

Completed Jobs (10)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
9	foreach at WriteIndexedRDD.scala:231	2016/10/06 00:47:57	0.2 s	1/1 (7 skipped)	7/7 (10162 skipped)
8	count at WriteIndexedRDD.scala:332	2016/10/06 00:47:56	1 s	1/1 (7 skipped)	7/7 (10162 skipped)
7	collect at HoodieClient.java:305	2016/10/06 00:46:41	1.2 min	2/2 (6 skipped)	2007/2007 (8162 skipped)
6	countByKey at WorkloadProfile.java:50	2016/10/06 00:46:29	11 s	4/4 (4 skipped)	8000/8000 (4162 skipped)
5	count at HoodieBloomIndex.java:74	2016/10/06 00:45:09	1.3 min	2/2 (3 skipped)	4000/4000 (2162 skipped)
4	sortByKey at HoodieBloomIndex.java:332	2016/10/06 00:45:08	1 s	1/1 (3 skipped)	2/2 (2162 skipped)
3	sortByKey at HoodieBloomIndex.java:332	2016/10/06 00:44:52	15 s	3/3 (1 skipped)	4005/4005 (157 skipped)
2	countByKey at HoodieBloomIndex.java:143	2016/10/06 00:44:47	5 s	2/2	10/10
1	countByKey at HoodieBloomIndex.java:137	2016/10/06 00:43:29	1.3 min	3/3	4157/4157
0	collect at IncrementalIngestor.scala:158	2016/10/06 00:43:19	6 s	1/1	100/100

Actual Writing of data

Index Lookup to identify location of record

Storage & Index

Implementation choices

Storage RDD

Every columnar file has one or more "redo" log

- **Row based Log Format - Apache Avro**
  - Append block
  - Rollback
- **Columnar Format - Apache Parquet**
  - Predicate pushdown
  - Columnar compression
  - Vectorized Reading

Index RDD - Insert vs Update during Ingest

- **Embedded**
  - Bloom Filter
- **External**
  - Key Value store

source

Correctness

Implementation choices

- **Commit File on DFS**
  - Atomic rename to publish data
  - Consume from downstream Spark job
- **Query Isolation**
  - Multiple versions of data file
  - Query hook - InputFormat via SparkSQL

source

Compaction

Implementation

- **Compaction**
  - Background Spark job
  - Lock log files
  - Minor
    - IO Bound Strategy
    - Improve Query Performance
  - Major
    - No log left behind

source

How can I query?

Show me the code !!!

```
SparkSession spark = SparkSession.builder()
                  .appName("Hoodie SparkSQL")
                  .config("spark.sql.hive.convertMetastoreParquet", false)
                  . enableHiveSupport()
                  .getOrCreate();

// real time query
spark.sql("select fare, begin_lon, begin_lat, timestamp from  hoodie.trips_rt where fare
> 100.0").show();

// read optimized query
spark.sql("select fare, begin_lon, begin_lat, timestamp from  hoodie.trips_ro where fare
> 100.0").show();

// Spark Datasource (WIP)
Dataset<Row> dataset = sqlContext.read().format(HOODIE_SOURCE_NAME)
    .option("query", "SELECT driverUUID, riderUUID FROM trips").load();
```

Query

Design choices

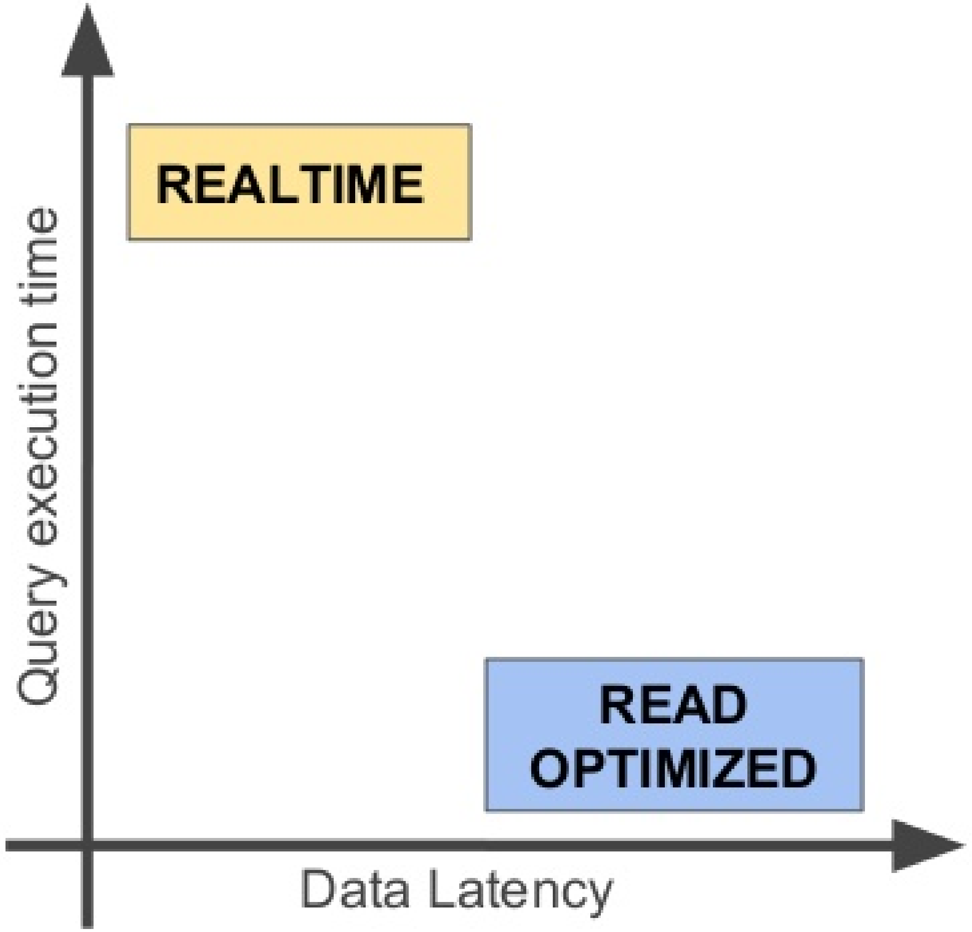
RUM Conjecture - Moving within the chosen area

Read Optimized View

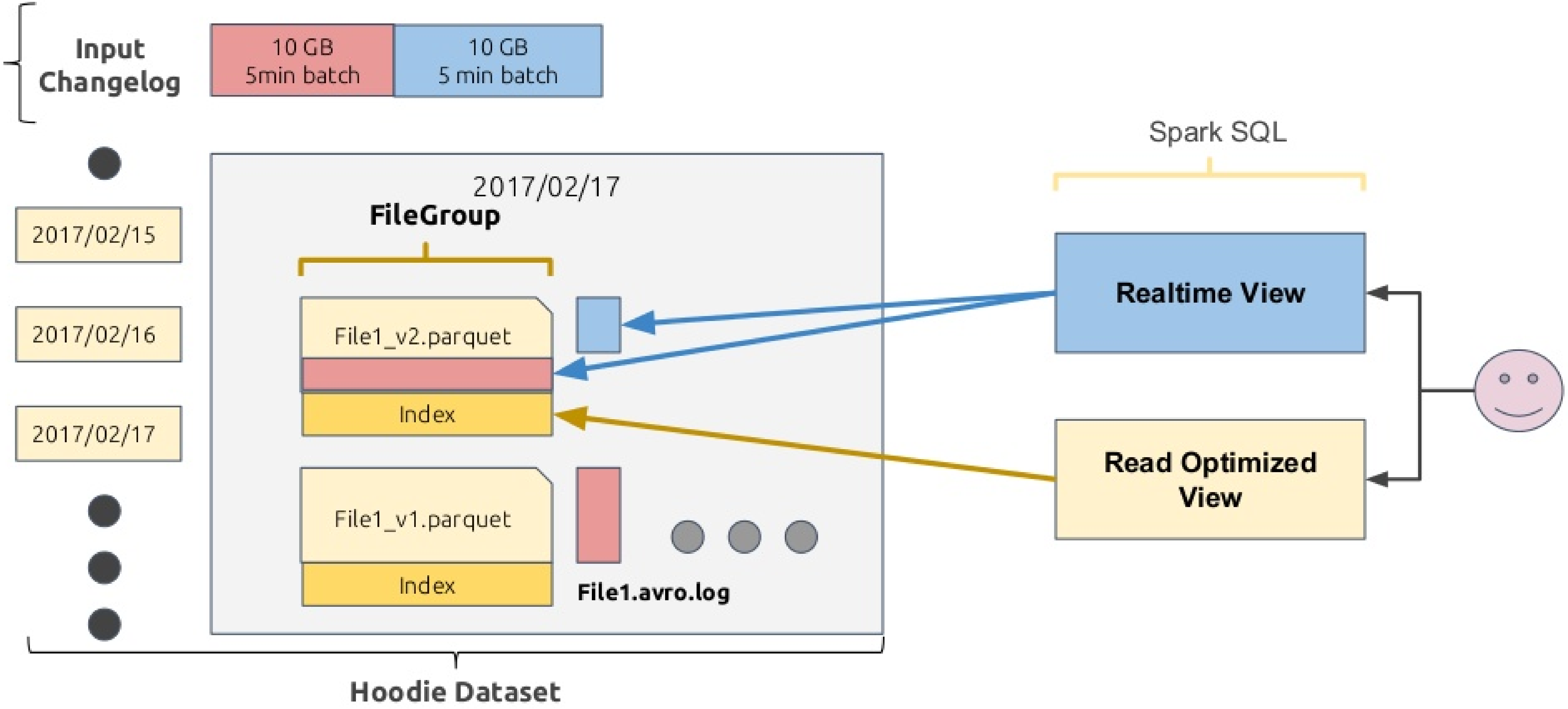
- Pick only columnar files for querying
- Raw Parquet Query Performance
- Freshness of Major Compaction

Real Time View

- Hybrid of row and columnar data
- Brings near-real time tables
- SparkSQL with convertMetaStore=false



Under The Hood



Community

Share love and code

Shopify evaluating for use

- Incremental DB ingestion onto GCS
- Early interest from multiple companies

Engage with us on Github (uber/hoodie)

- Look for “beginner-task” tagged issues
- Try out tools & utilities

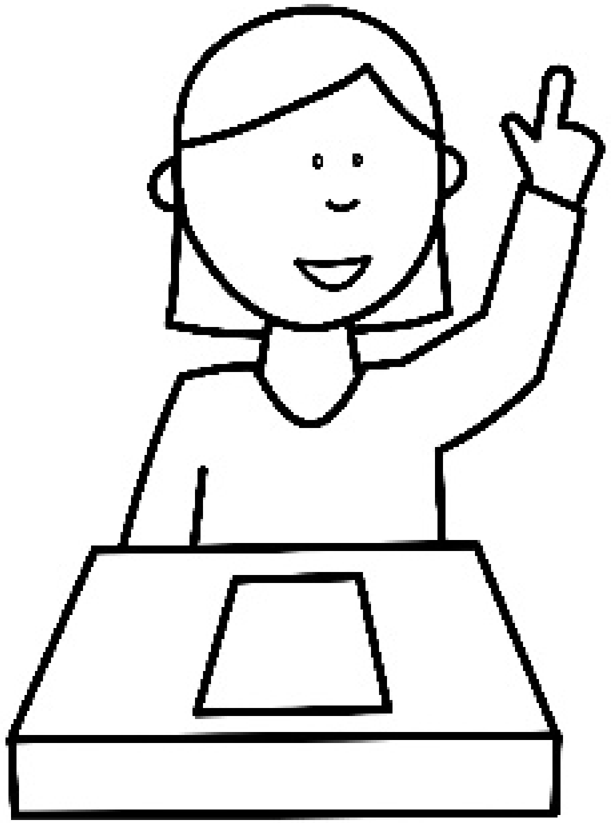
Uber is hiring for “Hoodie”

- Staff Engineer

Future Plans

Aim high

- Productionizing on AWS S3/EFS, GCP
- Spark Datasource
- Structured Streaming Sink
- Performance in Read Path
  - Presto plugin
  - Impala
- Spark Caching and integration with Apache Arrow
- Beam Runner



Questions?