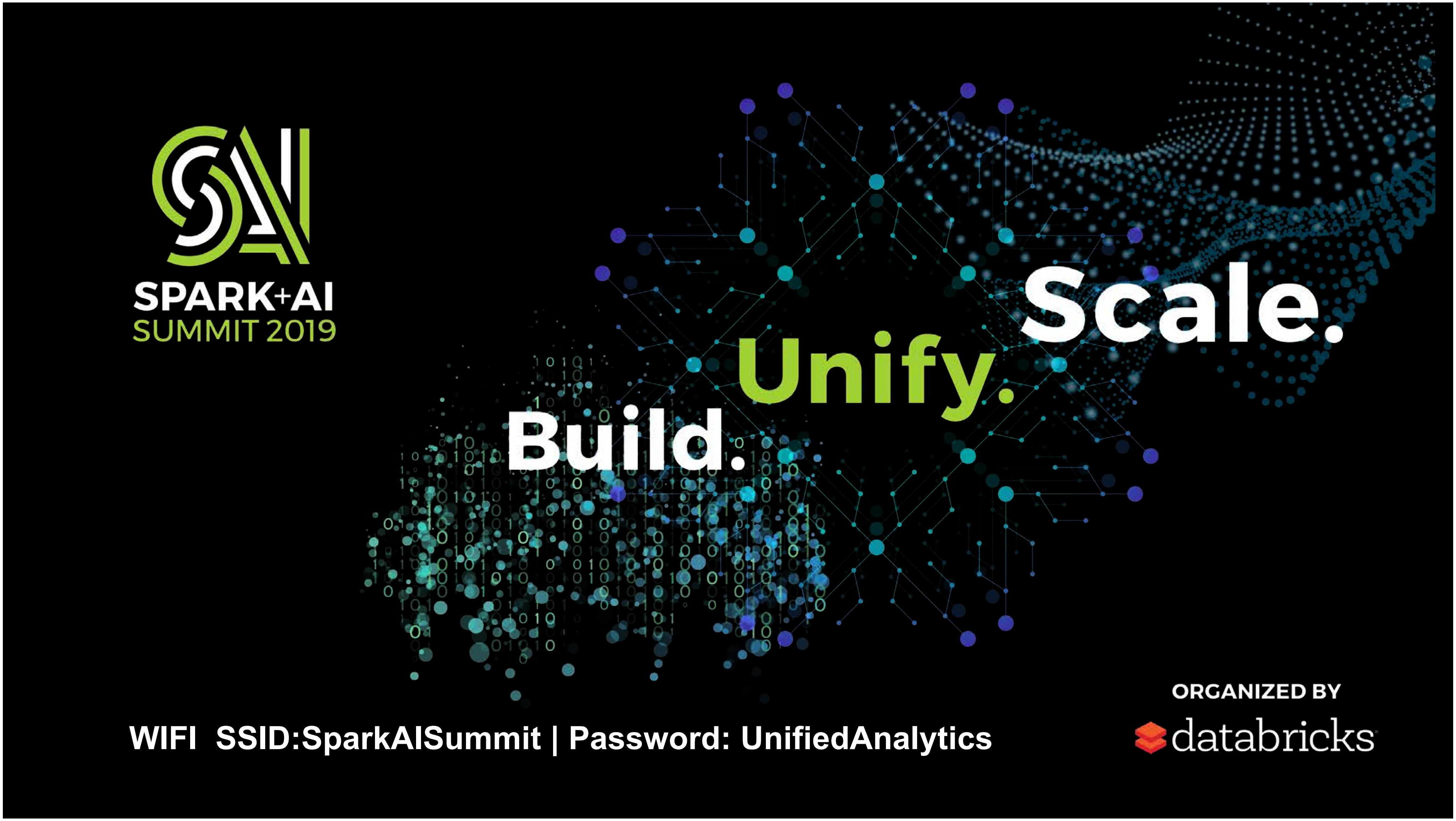

SPARK+AI
SUMMIT 2019
Scale.
Unify.
Build.
WIFI SSID:SparkAISummit | Password: UnifiedAnalytics
ORGANIZED BY
databricks

Near Real-Time Analytics with Apache Spark

Brandon Hamric, Data Engineer at Eventbrite
Beck Cronin-Dixon, Data Engineer at Eventbrite

#UnifiedAnalytics #SparkAISummit

Who are we

Brandon Hamric
Principal Data Engineer
Eventbrite

Beck Cronin-Dixon
Senior Data Engineer
Eventbrite

Overview

- Components of near real-time data
- Requirements
- Data ingestion approaches
- Benchmarks
- Considerations

Components of Near Real-time Data

- **Data**
  - Size
  - Mutability
  - Schema evolution
- **Deduplication**
- **Infrastructure**
  - Ad-hoc queries
  - Batch
  - Streaming queries

Components of Near Real-time Data

- **Storage**
  - Folder/file partitioning and bucketing
  - Format
- **Compression**
  - at-rest
  - in-transit

Common Requirements

- Latency is specific to use case
- Most stakeholders ask for latency in minutes
- Latency of dependencies*
- Some implementations fit certain latency better than others
- Mixed grain*

Milliseconds	Real-time services, Real-time ML prediction
Seconds	Up-to-date data dumps, transactional level reports, Fraud
Minutes	Up-to-date business metrics, Site health
Hours	Hourly Aggregations, Business health, Auditing
Days	Daily Aggregations, ML training, Accounting

Eventbrite Requirements

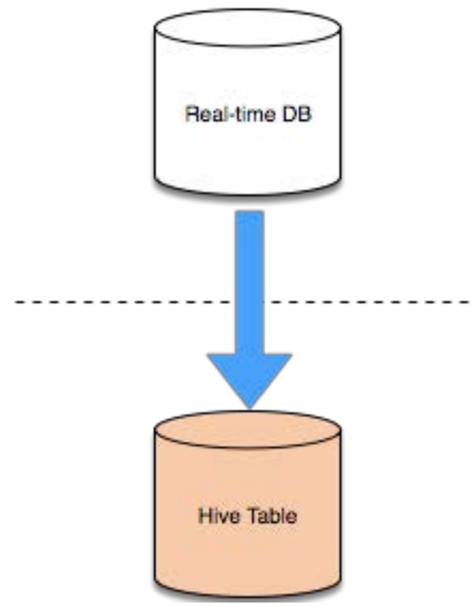
- Web interaction data: billions of records per month
- Real-time DB: Multiple TB
- Event Organizer reporting requirements
- Business and accounting reporting
- Mixed Grain - Seconds to Daily
- Stack: Spark, Presto, S3, HDFS, Kafka
- Parquet

Ingestion Approaches

- Full overwrite
- Batch incremental merge
- Append only
- Key/value store
- Hybrid batch/stream view

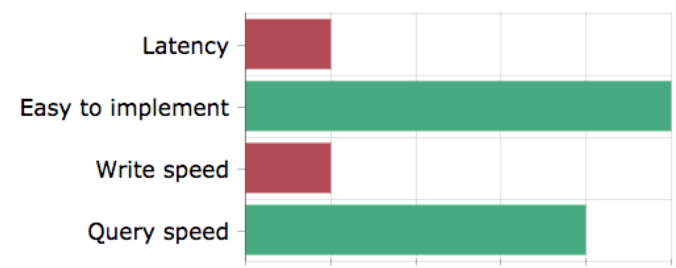
Approaches: Full Overwrite

- Batch Spark process
- Overwrite entire table every run
- Complete copy of real-time DB
- Direct ETL



Approaches: Full Overwrite

- **The Good**
  - Simple to implement
  - Ad Hoc Queries are fast/simple
- **The Bad**
  - Significant load on real-time DB
  - High Latency
  - High write I/O requirement

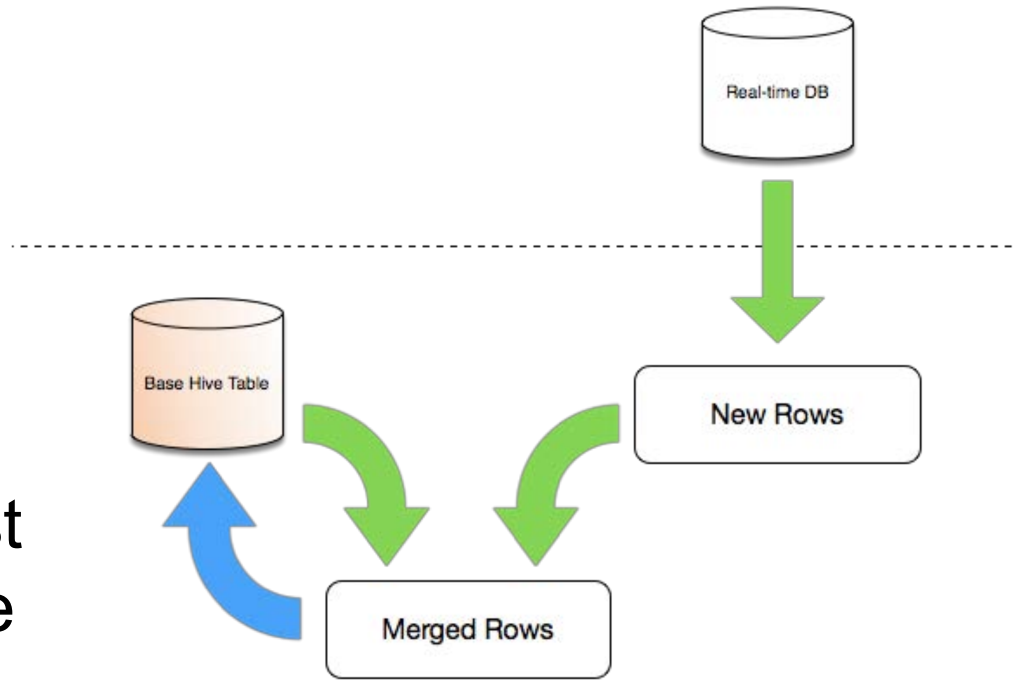


Full Overwrite Logic

```
events_jdbc_df = get_events_jdbc_dataframe_reader(numPartitions=240) \
    .option("partitionColumn", "id") \
    .option("lowerBound", min_changed_in_mysql) \
    .option("upperBound", max_changed_in_mysql) \
    .load()

events_jdbc_df.write.parquet("hdfs://user/warehouse/events")
```

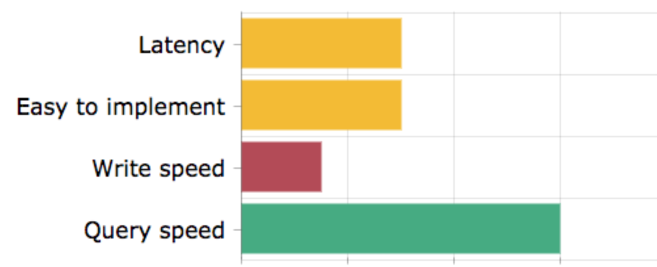
Approaches: Batch Incremental Merge

- Get new/changed rows
- Union new rows to base table
- Deduplicate to get latest rows
- Overwrite entire table
- limited latency to how fast you can write to data lake



Approaches: Batch Incremental Merge

- **The Good**
  - Lower load on real-time DB
  - Queries are fast/simple
- **The Bad**
  - Relatively high Latency
  - High write I/O requirement
  - Requires reliable incremental fields

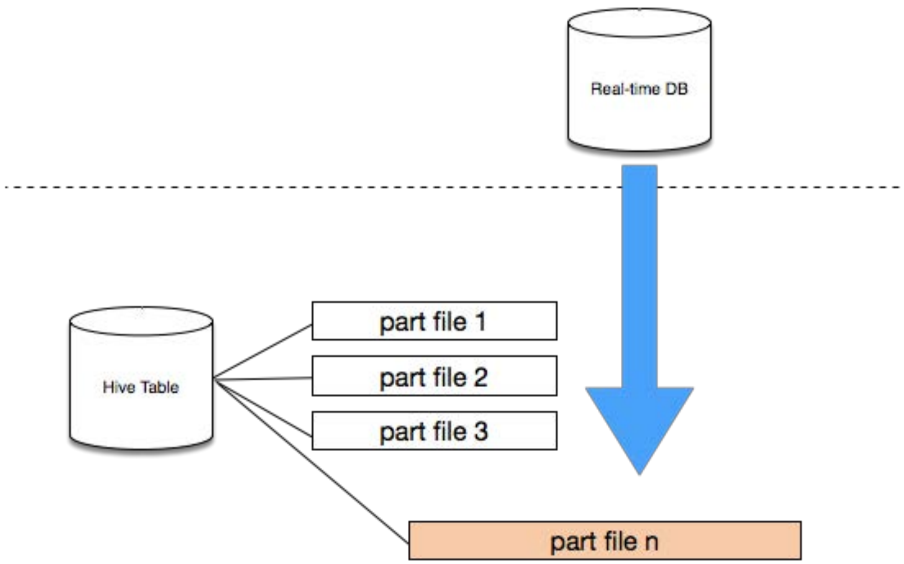


Batch Incremental Merge Logic

```
base_events_df = spark.read.parquet("hdfs://user/warehouse/demo/events")

# get new rows from mysql
new_mysql_events_df = get_events_jdbc_dataframe_reader(numPartitions=40) \
    .option("partitionColumn", "changed") \
    .option("lowerBound", max_changed_from_base) \
    .option("upperBound", max_changed_from_mysql) \
    .load() \
    .filter(F.col('changed') >= F.lit(max_changed_from_base))

# merge the new rows to the base table
updated_df = base_events_df \
    .union(new_mysql_events_df) \
    .orderBy("id", F.col("changed").desc()) \
    .dropDuplicates(["id"])

updated_df.write.parquet("hdfs://user/warehouse/demo/events")
```

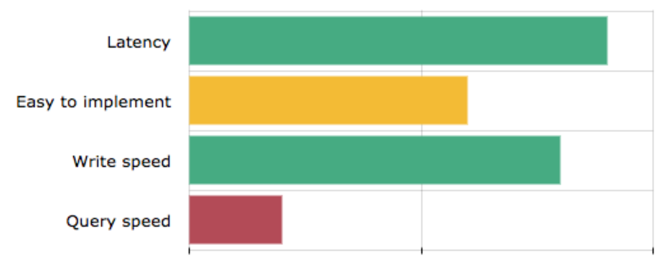
Approaches：Append-Only Tables

- Query the real-time db for new/changed rows
- Coalesce and write new part files
- Run compaction hourly, then daily



Approaches: Append Only

- **The Good**
  - Latency in minutes
  - Ingestion is easy to implement
  - Simplifies the data lake
  - Great for immutable source tables
- **The Bad**
  - Requires a compaction process
  - Extra logic in the queries
  - May require views

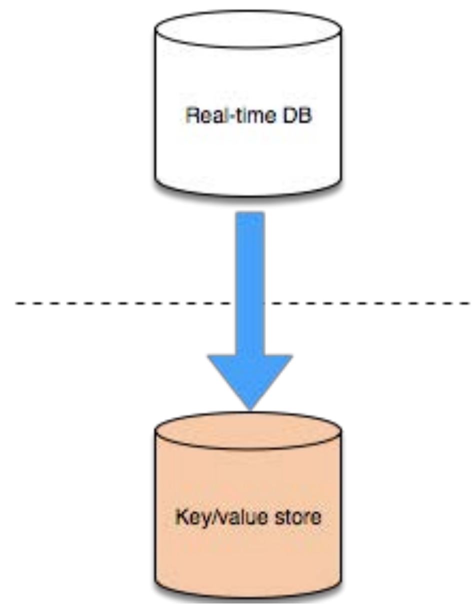


Append-Only Logic

```
base_events_df = spark.read.parquet("hdfs://user/warehouse/demo/events_append")

# get new rows from mysql
new_mysql_events_df = get_events_jdbc_dataframe_reader(numPartitions=40) \
    .option("partitionColumn", "changed") \
    .option("lowerBound", max_changed_from_base) \
    .option("upperBound", max_changed_from_mysql) \
    .load() \
    .filter(F.col("changed") >= F.lit(max_changed_from_base)) \
    .coalesce(1)

new_mysql_events_df \
    .write \
    .mode("append") \
    .parquet("hdfs://user/warehouse/demo/events_append")
```

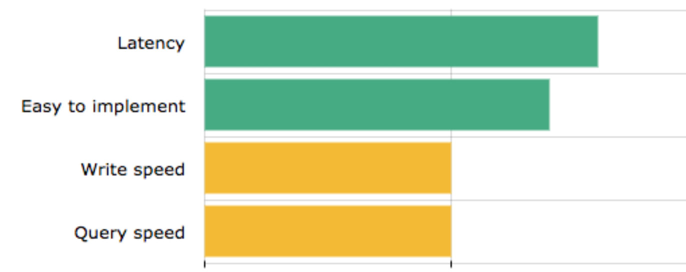
Approaches: Key/Value Store

- Query the real-time db for new/changed rows
- Upsert rows



Approaches: Key/Value Store

- **The Good**
  - Straightforward to implement
  - Built in idempotency
  - Good bridge between data lake and web services
- **The Bad**
  - Batch writes to a key/value store are slower than using HDFS
  - Not optimized for large scans

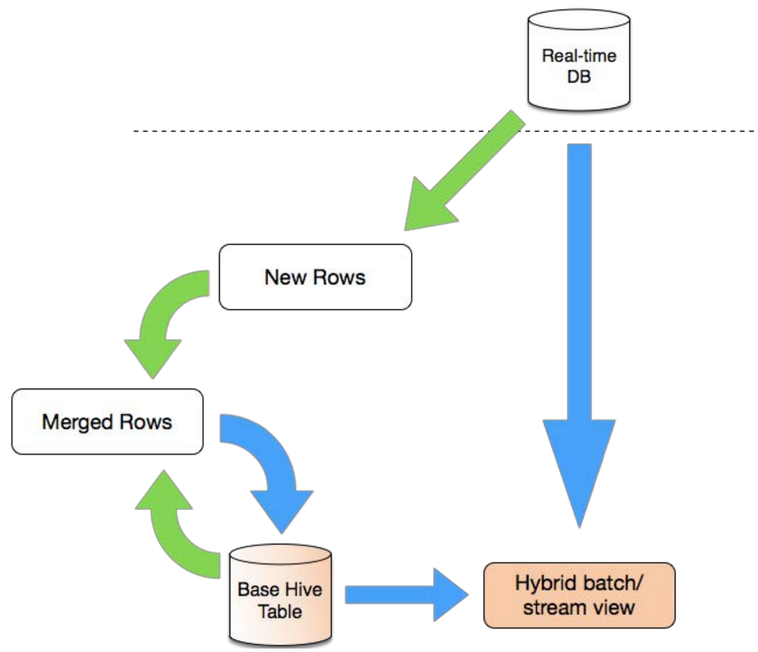


Key/Value Store Logic

```
# get new rows from mysql
new_mysql_events_df = get_events_jdbc_dataframe_reader(numPartitions=40) \
    .option("partitionColumn", "changed") \
    .option("lowerBound", max_changed_in_hdfs) \
    .option("upperBound", max_changed_in_mysql) \
    .load() \
    .filter(F.col("changed") >= F.lit(max_changed_in_hdfs))

# write the new rows to cassandra
new_mysql_events_df \
    .write \
    .format("org.apache.spark.sql.cassandra") \
    .mode("append") \
    .options(table="events", keyspace="demo") \
    .save()
```

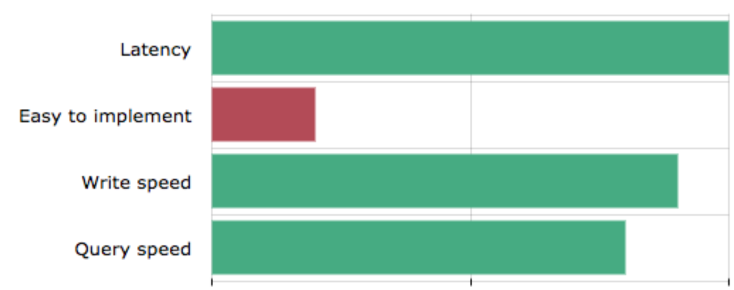
Approaches: Hybrid Batch/Stream View

- Batch ingest from DB transaction logs in kafka
- Batch merge new rows to base rows
- Store transaction ID in the base table
- Ad-hoc queries merge base table and latest transactions and deduplicate on-the-fly



Approaches: Hybrid Batch/Stream View

- **The Good**
  - Data within seconds*
  - Batch job is relatively easy to implement
  - Spark can do both tasks
- **The Bad**
  - Streaming merge is complex
  - Processing required on read

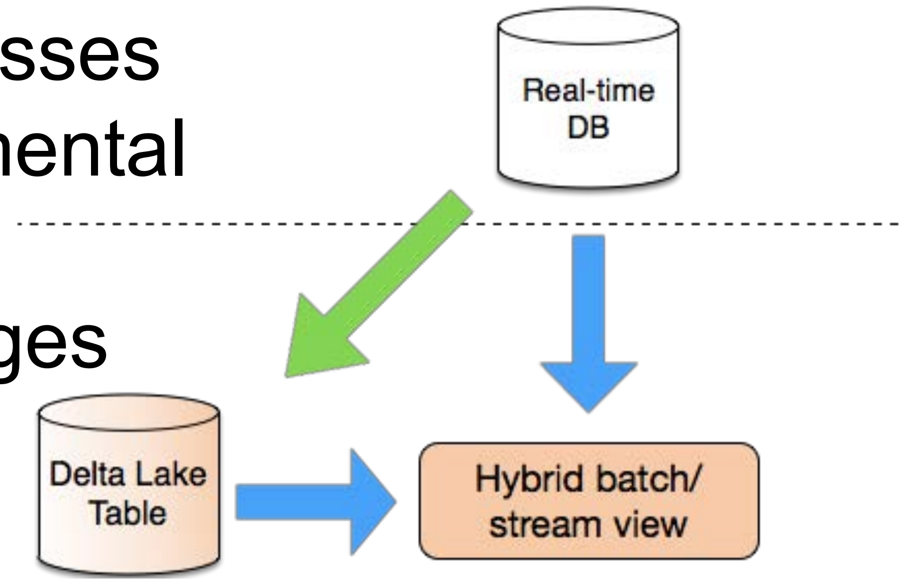


Hybrid Batch/Stream View

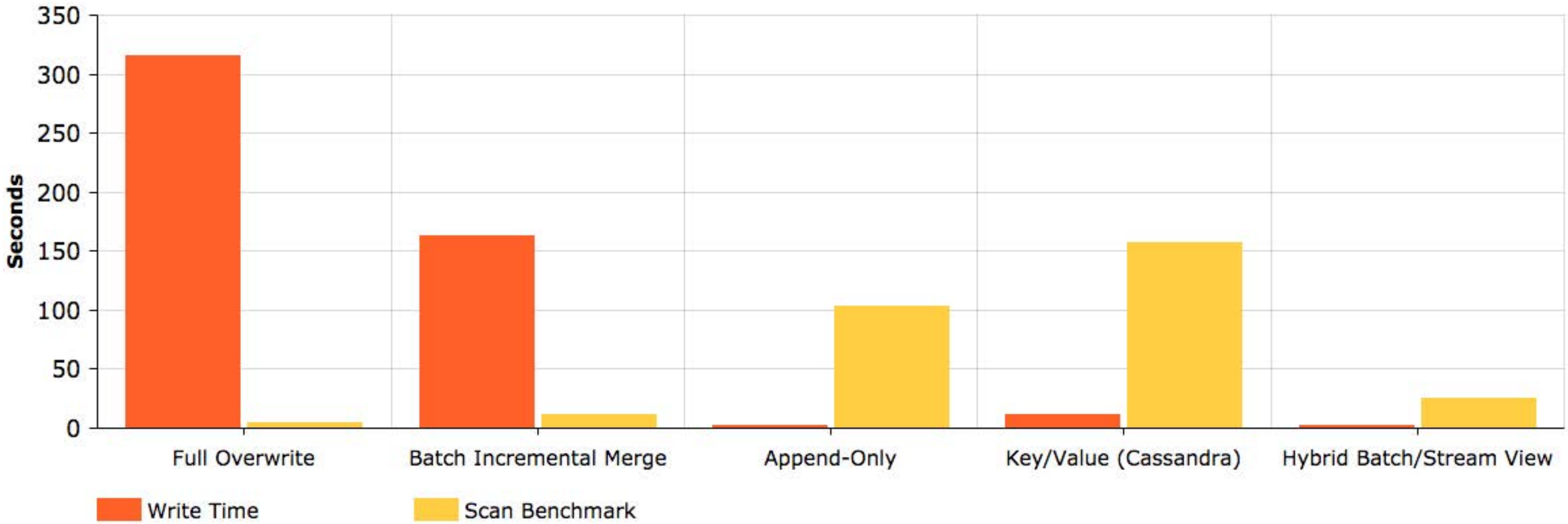
```
base_events_df = spark.read.parquet("hdfs://user/warehouse/events")

# get new rows from mysql
new_mysql_events_df = get_events_jdbc_reader(numPartitions=40) \
    .option("partitionColumn", "changed") \
    .option("lowerBound", max_changed_in_hdfs) \
    .option("upperBound", max_changed_in_mysql) \
    .load() \
    .filter(F.col("changed") >= F.lit(max_changed_in_hdfs))

# merge the base table and stream, then deduplicate
base_events_df \
    .union(new_mysql_events_df) \
    .orderBy("id", F.col("changed").desc()) \
    .dropDuplicates(["id"]) \
    .createOrReplaceTempView("events_live")

spark.sql("select ..... from events_live")
```

So Delta Lake is open source now...

- ACID transactions in Spark!!!!!!
- Frequent Ingestion Processes
- Simpler than other incremental merge approaches
- Hybrid approach still bridges latency gap



Read/Write Benchmark



Where is Spark Streaming?

- More frequent incremental writes
- Less stream to stream joins
- Decreasing batch intervals gives us more stream to batch joins
- Less stream to stream joins, means less memory and faster joins

Considerations

- Use case
- Latency needs
- Data size
- Deduplication
- Storage

sample mysql jdbc reader

```
def get_events_jdbc_reader(numPartitions):
    return spark.read.format("jdbc") \
        .option("url", "jdbc:mysql://demo.mysql.internal:3306/demo") \
        .option("driver", "com.mysql.jdbc.Driver") \
        .option("user", "spark") \
        .option("password", "password") \
        .option("dbTable", "events") \
        .option("numPartitions", numPartitions) \
        .option("fetchSize", 5000)
```

SPARK+AI
SUMMIT 2019

DON'T FORGET TO RATE AND REVIEW THE SESSIONS

SEARCH SPARK + AI SUMMIT

The picture can't be displayed.

SPARK+AI