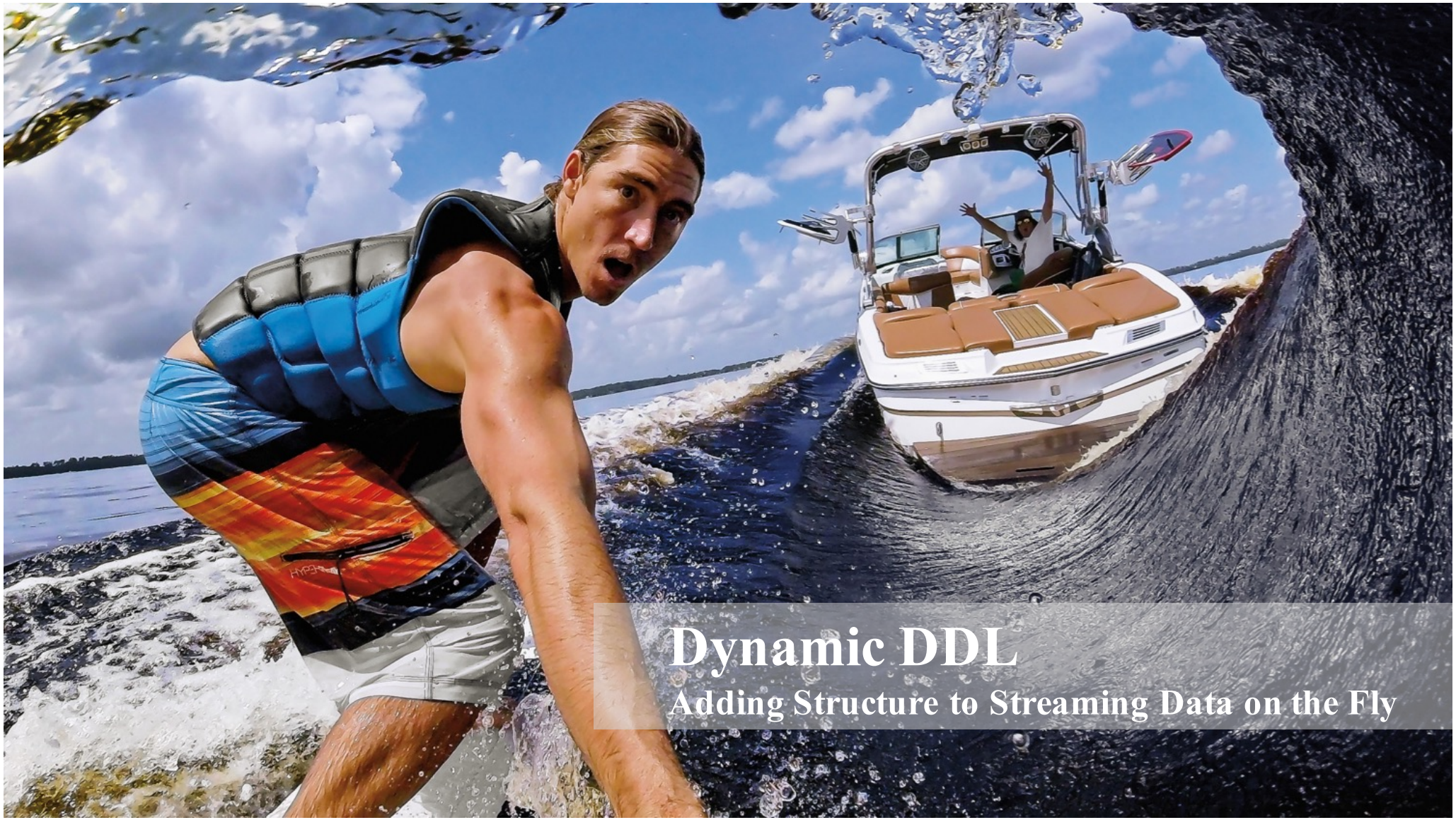


Dynamic DDL

Adding Structure to Streaming Data on the Fly

OUR SPEAKERS

David Winters

Big Data Architect
Data Science & Engineering
GoPro

Hao Zou

Software Engineer
Data Science & Engineering
GoPro

TOPICS TO COVER

- Background and Business
- GoPro Data Platform Architecture
  - Old File-based Pipeline Architecture
  - New Dynamic DDL Architecture
- Dynamic DDL Deep Dive
- Using Cloud-Based Services (Optional)
- Questions

Background and Business

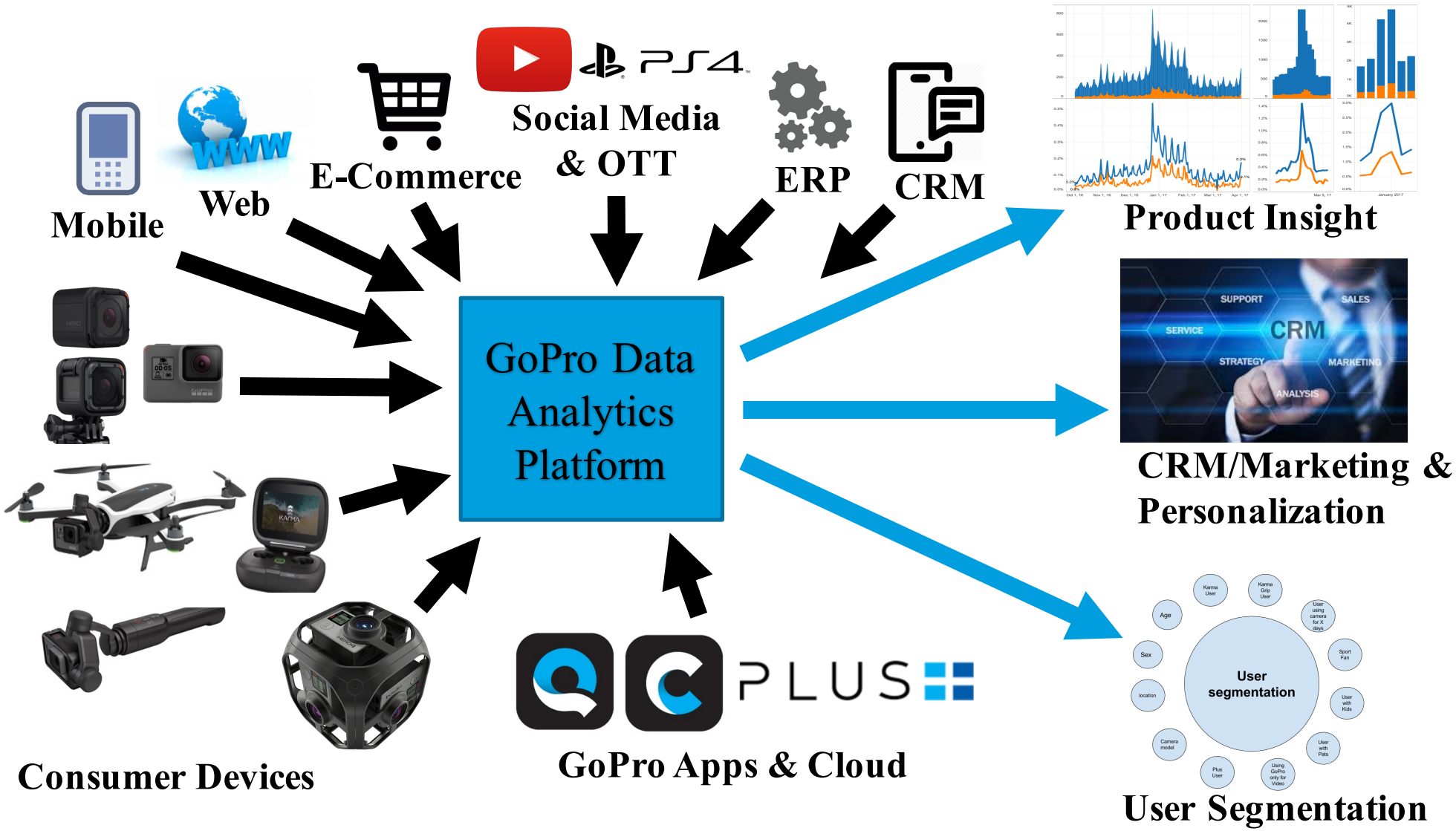
WHEN WE GOT HERE…

DATA ANALYTICS WAS BASED ON WORD OF MOUTH (& THIS GUY)

TODAY'S BUSINESS

PLUS

DEVELOPER PROGRAM


Mobile
Web
E-Commerce
Social Media & OTT
ERP
CRM
GoPro Data Analytics Platform
Product Insight
SUPPORT
SALES
SERVICE
CRM
STRATEGY
MARKETING
ANALYSIS
CRM/Marketing & Personalization
User segmentation
User Segmentation
Consumer Devices
PLUS
GoPro Apps & Cloud

DATA CHALLENGES AT GOPRO

- Variety of data - Hardware and Software products
  - Software - Mobile and Desktop Apps
  - Hardware - Cameras, Drones, Controllers, Accessories, etc.
  - External - CRM, ERP, OTT, E-Commerce, Web, Social, etc.
- Variety of data ingestion mechanisms - Lambda Architecture
  - Real-time streaming pipeline - GoPro products
  - Batch pipeline - External 3rd party systems
- Complex Transformations
  - Data often stored in binary to conserve space in cameras
  - Heterogeneous data formats (JSON, XML, and packed binary)
- Seamless Data Aggregations
  - Blend data between different sources, hardware, and software
  - Build structures which reflect state vs. event-based
- Handle Privacy & Anonymization


Data Platform Architecture

OLD FILE-BASED PIPELINE ARCHITECTURE

Streaming

Real Time Cluster
- Log file streaming
- RESTful service
- Kafka
- Spark Streaming
- HBase

JSON

ETL Cluster
- Aggregations and Joins
- Hive and Spark jobs
- Map/Reduce
- Airflow

Parquet

DDL

Secure Data Mart Cluster
- End User Query
- Impala / Sentry
- Parquet
- Kerberos & LDAP

Analytics Apps
- Hue
- Tableau
- Plotly
- Python
- R

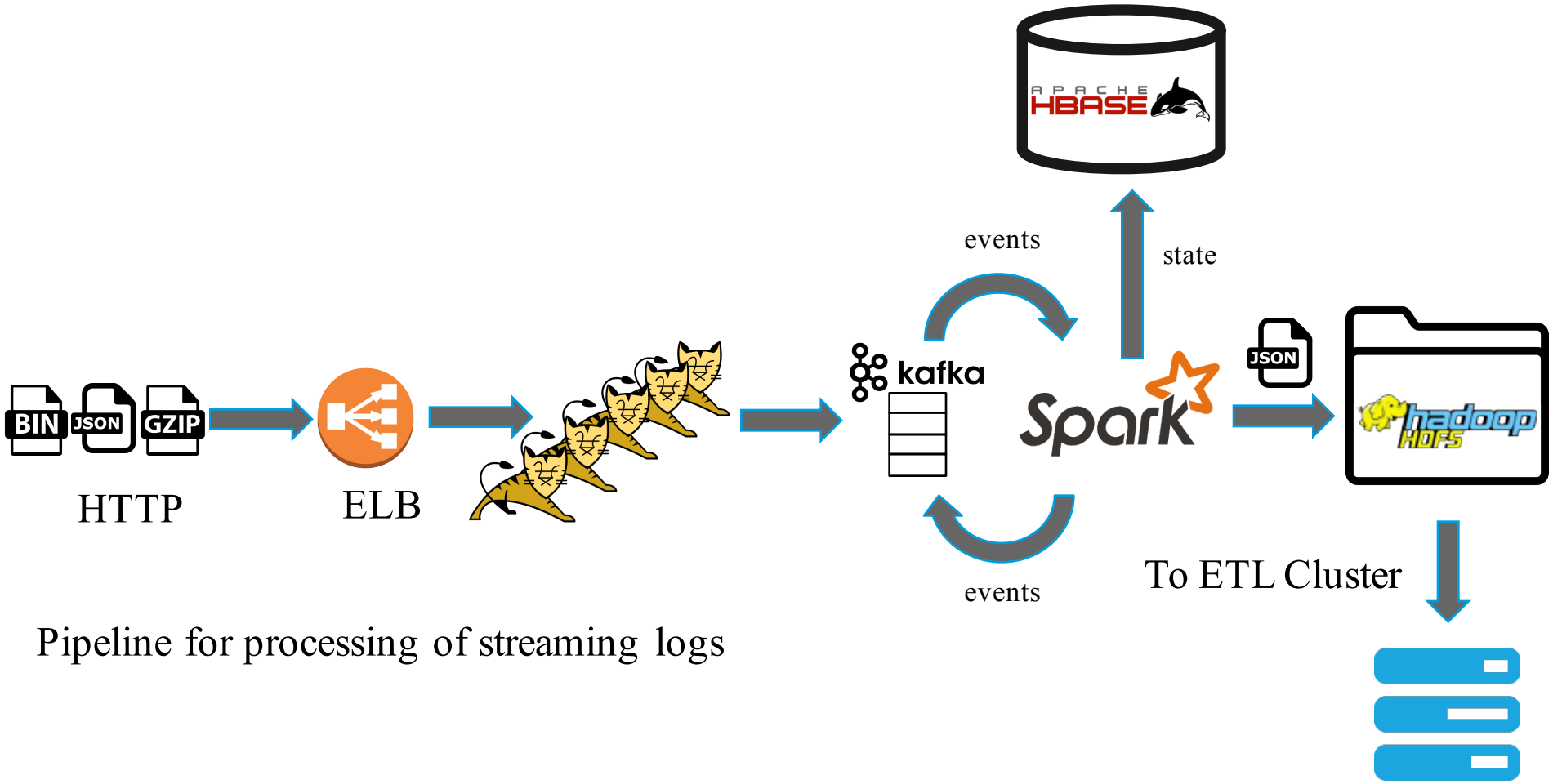
- Rest API
- FTP downloads
- S3 sync

Batch Download

Batch Induction Framework
- Batch files
- Scheduled downloads
- Pre-processing
- Java App
- Airflow

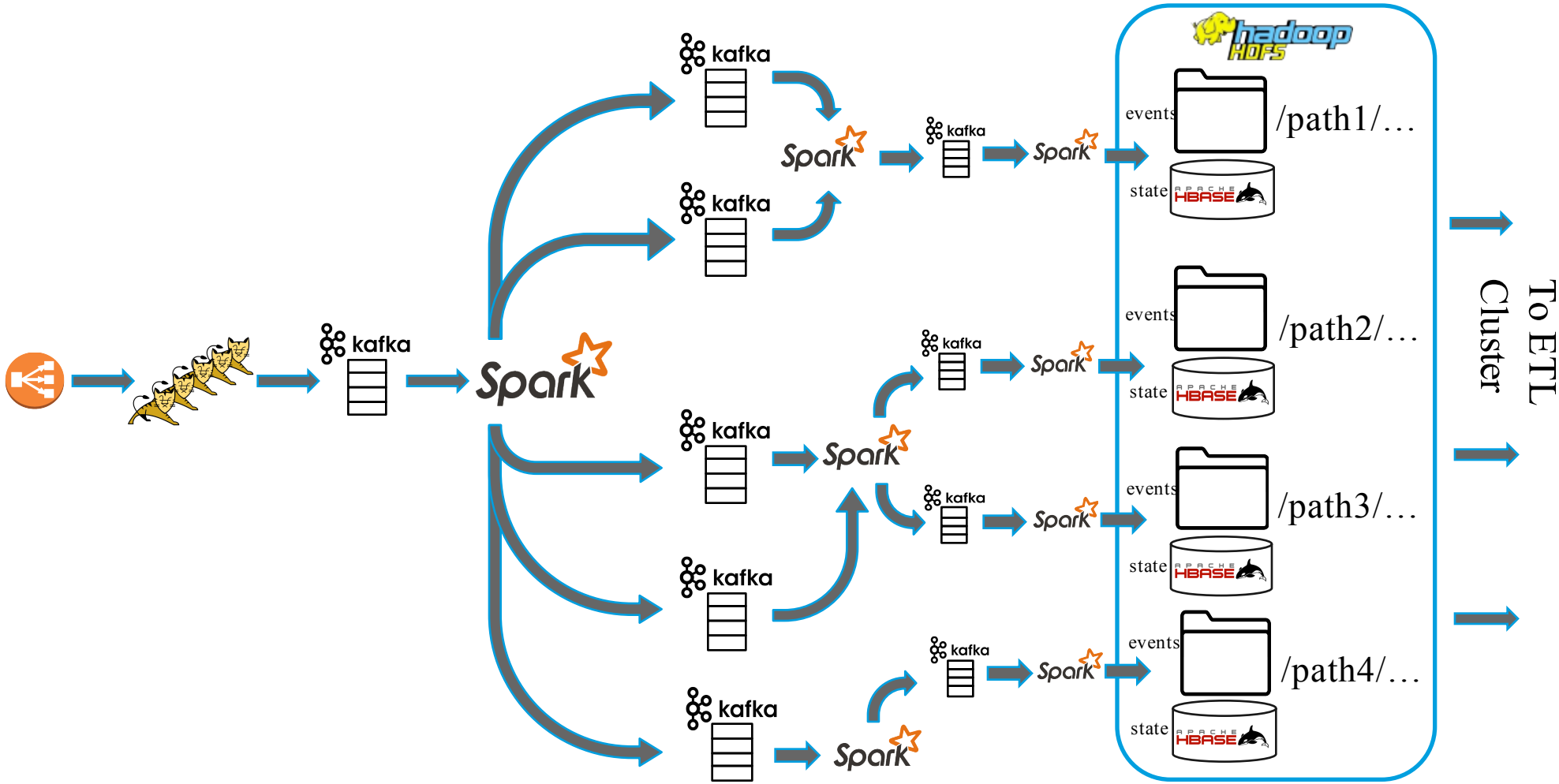
JSON

STREAMING ENDPOINT

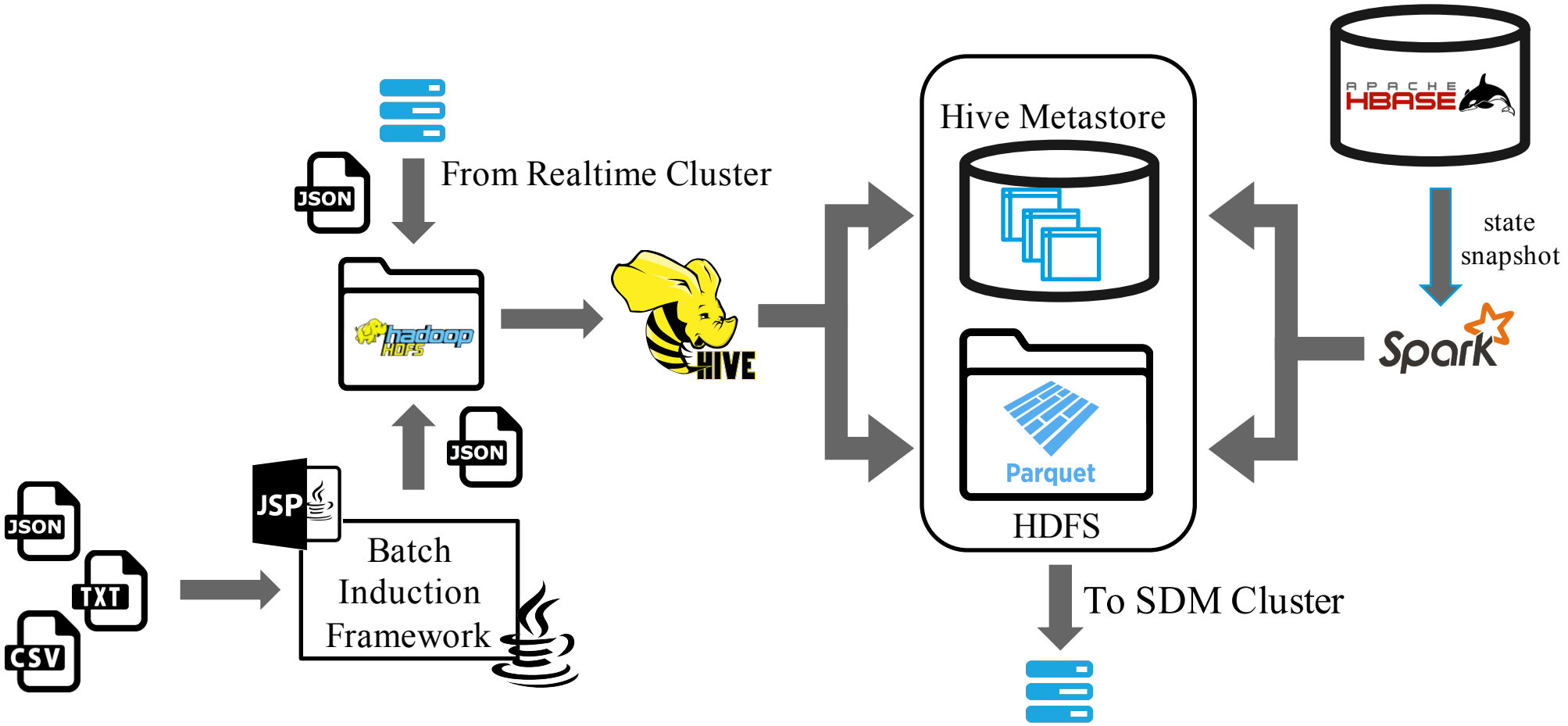


Pipeline for processing of streaming logs

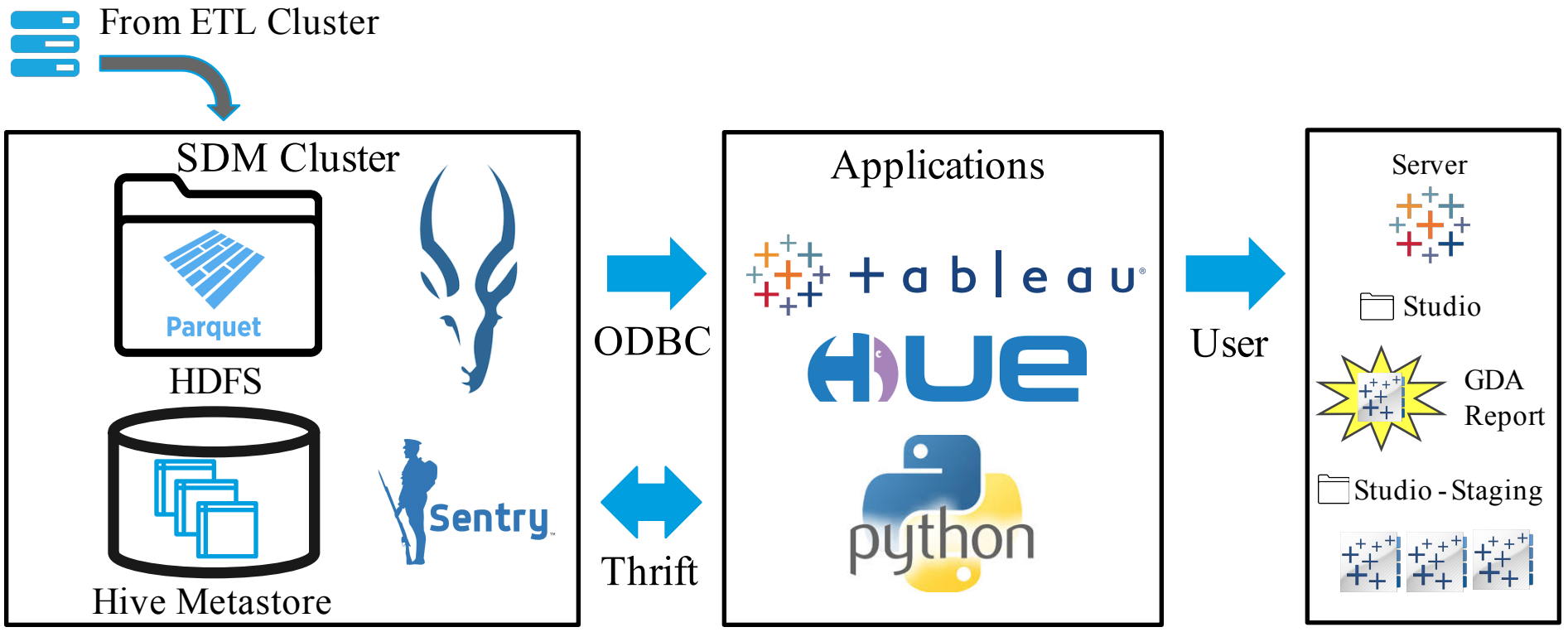
SPARK STREAMING PIPELINE



ETL PIPELINE



DATA DELIVERY!

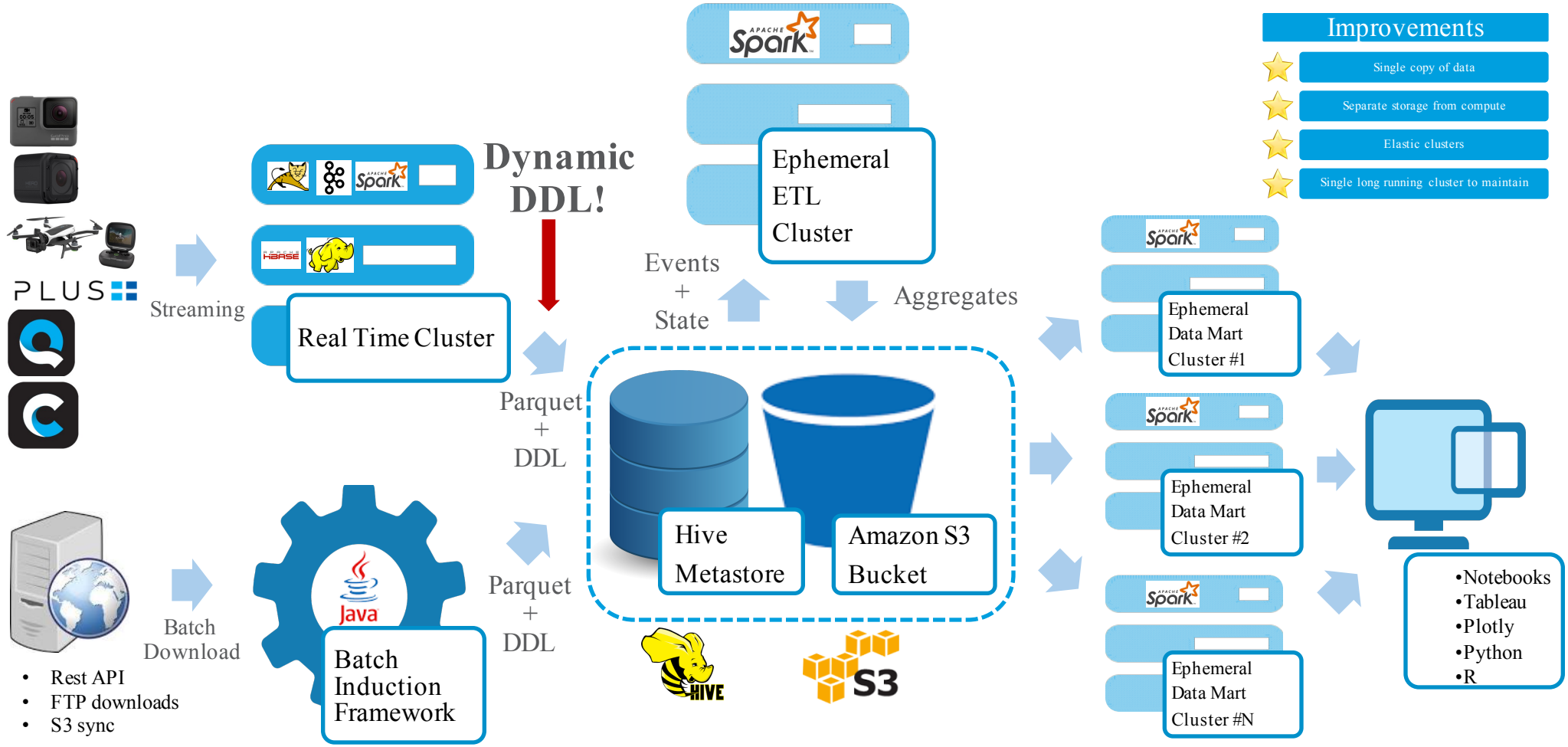


PROS AND CONS OF OLD SYSTEM

- Isolation of workloads
- Fast ingest
- Secure
- Fast delivery/queries
- Loosely coupled clusters

- Multiple copies of data
- Tightly coupled storage and compute
- Lack of elasticity
- Operational overhead of multiple clusters

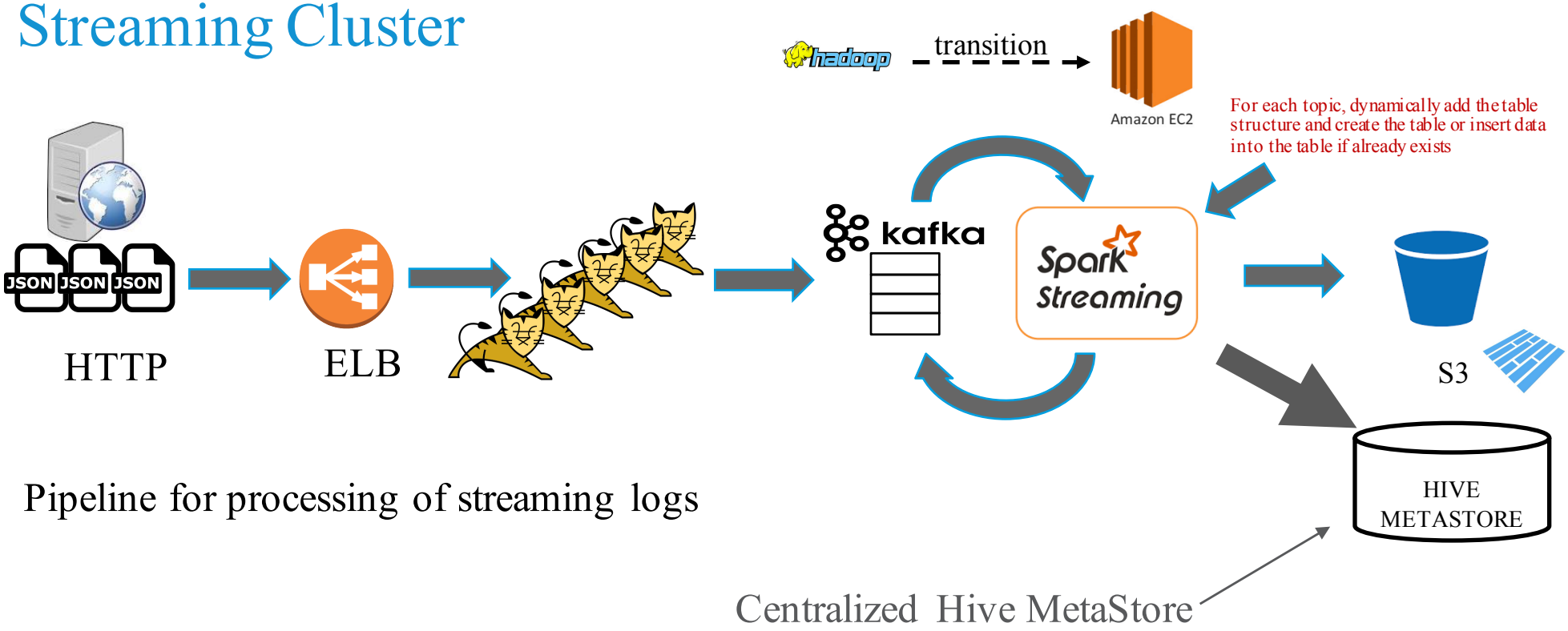
NEW DYNAMIC DDL ARCHITECTURE



Dynamic DDL Deep Dive

NEW DYNAMIC DDL ARCHITECTURE

Streaming Cluster



Pipeline for processing of streaming logs

DYNAMIC DDL

- What is Dynamic DDL?
    - Dynamic DDL is adding structure (schema) to the data on the fly whenever the providers of the data are changing their structure.
- Why is Dynamic DDL needed?
    - Providers of data are changing their structure constantly. Without Dynamic DDL, the table schema is hard coded and has to be manually updated based on the changes of the incoming data.
    - All of the aggregation SQL would have to be manually updated due to the schema change.
    - Faster turnaround for the data ingestion. Data can be ingested and made available within minutes (sometimes seconds).
- How we did this?
    - Using Spark SQL/Dataframe
    - See example

DYNAMIC DDL

- Example:

```
{"_data":{"record":{"id":  "1", "first_name":  "John", "last_name":  "Fork", "state": "California",  "city": "san Mateo"}, "log_ts":"2016-07-20T00:06:01Z"}}
```

Flatten the data first

```
{"record_key":"state","record_value":"California","id":"1","log_ts":"2016-07-20T00:06:01Z"}
{"record_key":"last_name","record_value":"Fork","id":"1","log_ts":"2016-07-20T00:06:01Z"}
{"record_key":"city","record_value":"san  Mateo","id":"1","log_ts":"2016-07-20T00:06:01Z"}
{"record_key":"first_name","record_value":"John","id":"1","log_ts":"2016-07-20T00:06:01Z"}
```

Dynamically generated schema

Fixed schema

id	log_ts	record_key	record_value
1	2016-07-20T00:06:01Z	state	California
1	2016-07-20T00:06:01Z	last_name	Fork
1	2016-07-20T00:06:01Z	city	san Mateo
1	2016-07-20T00:06:01Z	first_name	John

data_log_ts	data_record_city	data_record_first_name	data_record_id	data_record_last_name	data_record_state
2016-07-20T00:06:01Z	san Mateo	John	1	Fork	California

```
SELECT MAX(CASE WHEN record_key  = 'state' THEN record_value ELSE null END) AS data_record_state,
MAX(CASE WHEN record_key  = 'last_name' THEN record_value ELSE null END) As data_record_last_name,
MAX(CASE WHEN record_key  = 'first_name' THEN record_value ELSE null END) As data_record_first_name,
MAX(CASE WHEN record_key  = 'city' THEN record_value ELSE null END) AS data_record_city,
id as data_record_id,  log_ts as data_log_ts from test group by id, log_ts
```

DYNAMIC DDL USING SPARK SQL/DATAFRAME

- Code snippet of Dynamic DDL transforming new JSON attributes into relational columns

Add the partition columns

```
var partitionedEvents = addPartitionColumns(df, partitionColumnNames)
```

Manually create the table due to a bug in spark

```
sqlContext.sql(
  s"""CREATE TABLE IF NOT EXISTS $dbTableName
     |(${partitionedEvents.schema.fieldNames.filterNot(partitionColumnNames.contains(_)).map(_ + " STRING").mkString(", ")})
     |PARTITIONED BY (${partitionColumnNames.map(_ + " STRING").mkString(", ")})
     |STORED AS ${destFormat.split('.').last}
     |${if (destDir != null && destDir.startsWith("s3")) {s"LOCATION '$destDir/$destTable'"} else {""}}""".stripMargin)
```

DYNAMIC DDL USING SPARK SQL/DATAFRAME

Add the new columns that exist in the incoming data frame but do not exist yet in the destination table

```
var tableDf = sqlContext.table(dbTableName)

// Add any new columns that exist in the incoming data frame but do not exist yet in the destination table.
val newFields = partitionedEvents.schema.fieldNames.filter(f => {
  !tableDf.schema.fieldNames.contains(f)
})
if (!newFields.isEmpty) {
  if (LOG.isInfoEnabled) {
    LOG.info(s"""These fields do not exist in the Hive metastore for table '$dbTableName' and will be added: ${newFields.mkString(", ")}""")
  }
  sqlContext.sql(s"""ALTER TABLE $dbTableName ADD COLUMNS (${newFields.map(_ + " STRING").mkString(", ")})""")
  sqlContext.asInstanceOf[HiveContext].refreshTable(dbTableName)
  tableDf = sqlContext.table(dbTableName)
}
```

This syntax is not working anymore after upgrading to spark 2.x

DYNAMIC DDL USING SPARK SQL/DATAFRAME

Three temporary way to solve the problem in spark 2.x:

- Launch a hiveserver2 service, then use jdbc call hive to alter the table
- Use spark to directly connect to hivemetastore, then update the metadata
- Update spark source code to support Alter table syntax and repackage it

DYNAMIC DDL USING SPARK SQL/DATAFRAME

Project all columns from the table

```
// Reorder the columns in the incoming data frame to match the order in the destination table.
// Project all columns from the table.  For columns that are missing from the incoming data frame,
// project a null value aliased as the missing column name.
partitionedEvents = partitionedEvents.select(tableDf.schema.fieldNames.map(
  f => {
    if (partitionedEvents.columns.contains(f)) col(f) else lit(null).as(f)
  }): _*)
```

Append the data into the destination table

```
// Coalesce the data frame into the desired number of partitions (files).
val coalescedEvents = partitionedEvents.coalesce(destPartitions)
coalescedEvents.write.mode(SaveMode.Append).format(destFormat).partitionBy(partitionColumnNames: _*).saveAsTable(dbTableName)
```

DYNAMIC DDL USING SPARK SQL/DATAFRAME

- Reprocessing the DDL Table with new partition Key (Tuning tips)

Choose the partition key wisely

Use coalesce if there too many partitions

Use filter if Data still too large

Add the new partition key

```
val newPartitionedDF = originalDf
  .withColumn(partitionColumnNames.get(0), getDateStringFromMillisUdf($"epmeta_epts"))
  .withColumn(partitionColumnNames.get(1), getHourStringFromMillisUdf($"epmeta_epts"))
```

Use Coalesce to control the job tasks

```
newPartitionedDF.coalesce(numOfPartitions).write.mode(saveModeName)
  .partitionBy(partitionColumnNames.get(0), partitionColumnNames.get(1))
  .saveAsTable(outputInfo.tableName())
```

Using Cloud-based Services

USING S3: WHAT IS S3?

- S3 is not a file system.
- S3 is an object store. Similar to a key-value store.
- S3 objects are presented in a hierarchical view but are not stored in that manner.
    - S3 objects are stored with a key derived from a “path”.
        - The key is used to fan out the objects across shards.
        - The path is for display purposes only. Only the first 3 to 4 characters are used for sharding.
- S3 does not have strong transactional semantics but instead has eventual consistency.
    - S3 is not appropriate for realtime updates.
    - S3 is suited for longer term storage.

USING S3: BEHAVIORS

- S3 has similar behaviors to HDFS but even more extreme.
  - Larger latencies
  - Larger files/writes – Think GBs
- Write and read latencies are larger but the bandwidth is much larger with S3.
  - Thus throughput can be increased with parallel writers (same latency but more throughput through parallel operations)
    - Partition your RDDs/DataFrames and increase your workers/executors to optimize the parallelism.
- Each write/read has more overhead due to the web service calls.
  - So use larger buffers.
  - Match the size of your HDFS buffers/blocks if reading/writing from/to HDFS.
  - Collect data for longer durations before writing large buffers in parallel to S3.
- Retry logic – Writes to S3 can and will fail.
- Cannot stream to S3 – Complete files must be uploaded.
  - Technically, you can simulate streaming with multipart upload.

USING S3: TIPS

- Tips for using S3 with HDFS
  - Use the s3a scheme.
    - Many optimizations including buffering options (disk-based, on-heap, or off-heap) and incremental parallel uploads (S3A Fast Upload).
    - More here: http://hadoop.apache.org/docs/current/hadoop-aws/tools/hadoop-aws/index.html#S3A
  - Don't use rename/move.
    - Moves are great for HDFS to support better transactional semantics when streaming files.
    - For S3, moves/renames are copy and delete operations which can be very slow especially due to the eventual consistency.
- Other advanced S3 techniques:
  - Hash object names to better shard the objects in a bucket.
  - Use multiple buckets to increase bandwidth.

QUESTIONS?