DATA+AI SUMMIT
BY databricks

UBER'S BATCH ANALYTICS EVOLUTION FROM HIVE TO SPARK

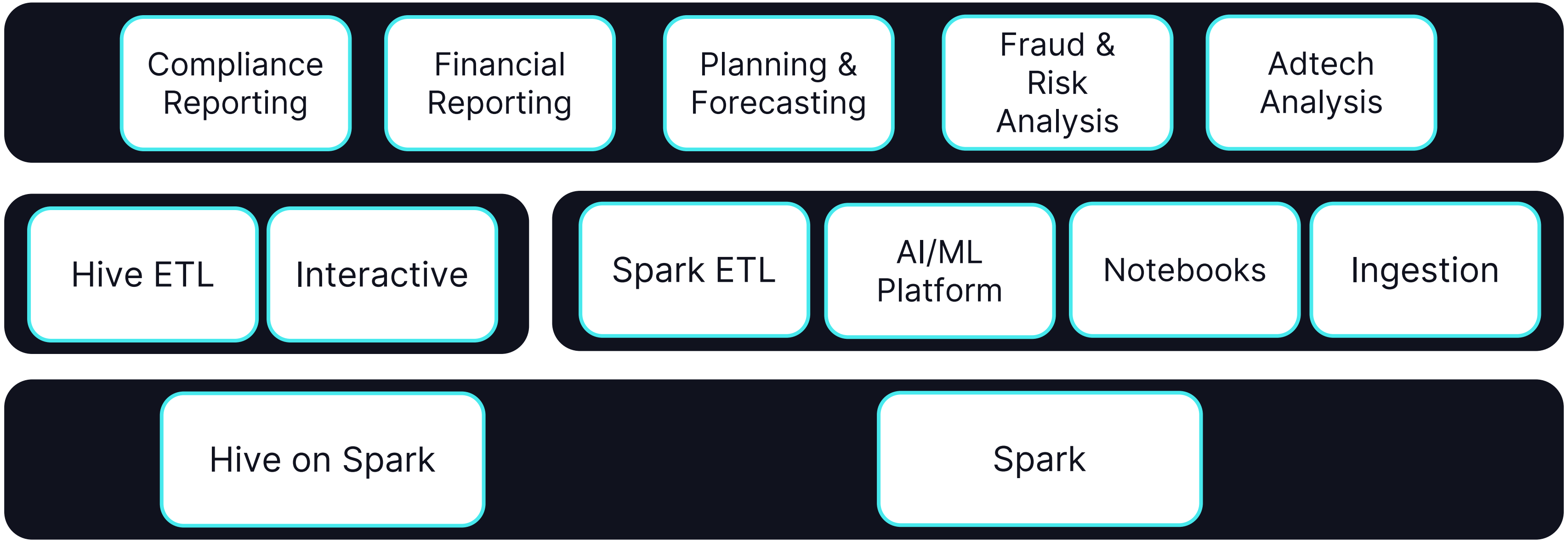
Kumudini, Akshayaprakash

Uber

Agenda

- Hive @ Uber
- Motivation To Migrate
- Migration Strategy
- Hive to SparkSQL Translation
- Shadow Testing & Data validation
- Hive-Spark Disparity
- Results & Future Work

Batch Analytics @ Uber



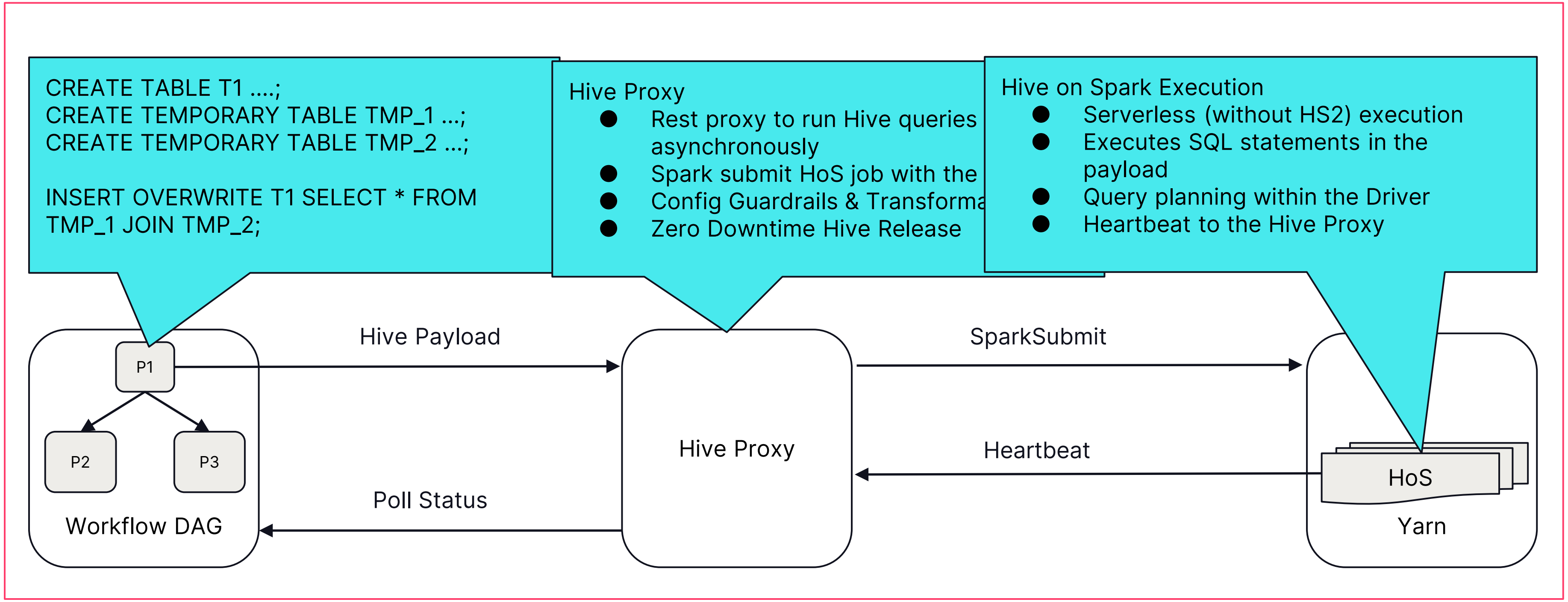
Hive @ Uber

18K	5M	150K	35%
Total ETLs	Monthly Scheduled Queries	Monthly Interactive Queries	Yarn Usage

Hive: 2.3
Spark: 2.4.3

Hive @ Uber

Architecture



```
CREATE TABLE T1 ....;
CREATE TEMPORARY TABLE TMP_1 ...;
CREATE TEMPORARY TABLE TMP_2 ...;

INSERT OVERWRITE T1 SELECT * FROM
TMP_1 JOIN TMP_2;
```

Hive Proxy
- Rest proxy to run Hive queries asynchronously
- Spark submit HoS job with the
- Config Guardrails & Transforma
- Zero Downtime Hive Release

Hive on Spark Execution
- Serverless (without HS2) execution
- Executes SQL statements in the payload
- Query planning within the Driver
- Heartbeat to the Hive Proxy

Motivation to Migrate

Hive 2.3 to Spark 3.3.2

Active OSS development

- Hive on Spark (HoS) has inactive OSS development, Obsolete!
- Spark3 has vibrant OSS community

Better Performance

- Hive has static query planning
- Adaptive Query Execution in Spark3
- Compute & Cost Efficiency

Unified Batch Analytics

- Single engine for all batch analytics use cases
- Uber’s observability & performance optimization tools already integrated well with Spark

Migration Strategy

2-Step Migration Process

Step 1: Automated Migration

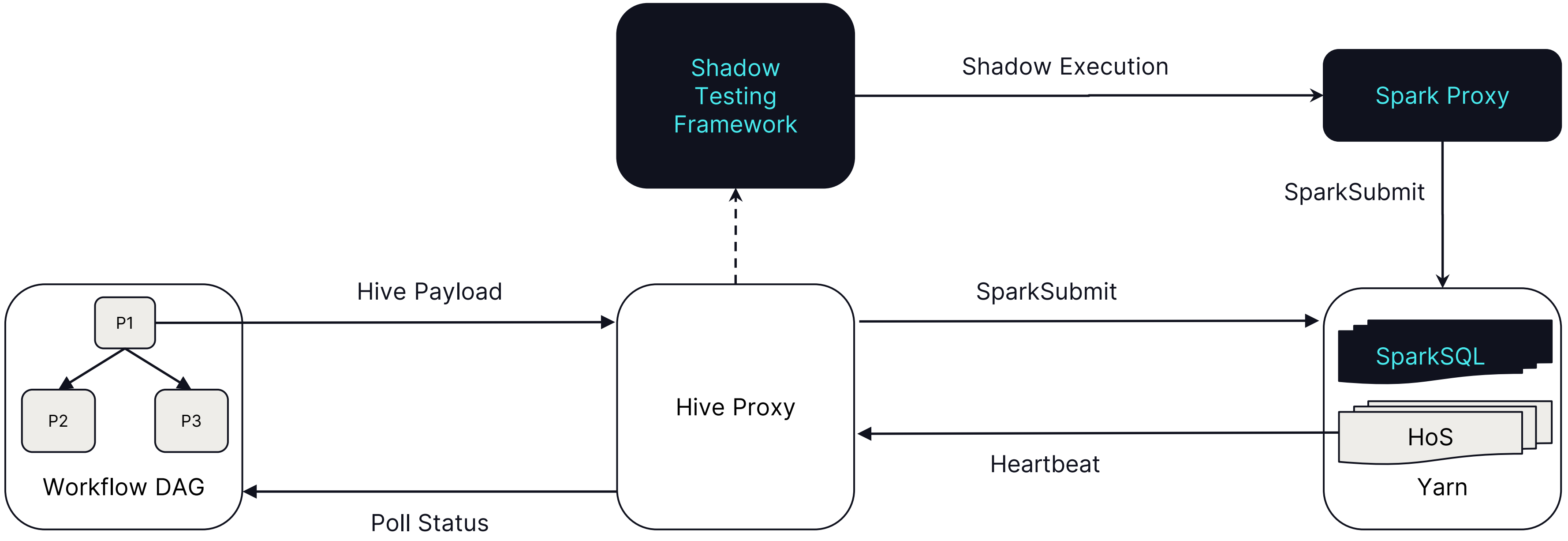
- Translate: Dynamic Translation of HiveQL to SparkSQL
- Validate: Shadow Testing of generated SparkSQL
- Migrate: Migration of HiveQL to SparkSQL

Step 2: Source Code Update

- Automation for updating static HQLs.
- Collaboration with workflow owners to update dynamically generated HQLs.

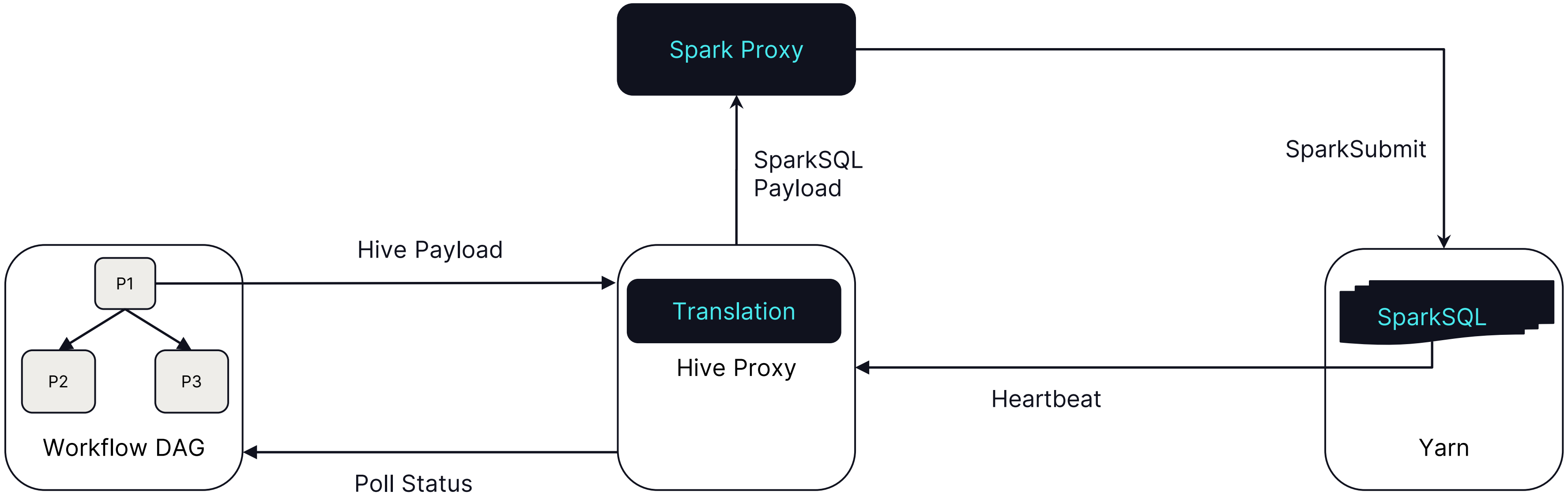
Automated Migration

Shadow Testing



Automated Migration

Migration



Hive to SparkSQL Translation

- Leveraged Coral's framework
- Added support for Hive2.x grammar
- Removed dependency on Calcite's RelNode
  - No query optimization
  - No semantic validation against HMS
- Added rules to support syntax like DDLs and unregistered UDFs.
- Added support for payload translation

Hive to SparkSQL Translation

```
SET hive.auto.convert.join=true;
SET hive.auto.convert.join.noconditionaltask=true;
SET hive.auto.convert.join.noconditionaltask.size=128000000;

SELECT
    max(struct(trip_count, city_id)).col2 as city_id
FROM
    trip_info;
```

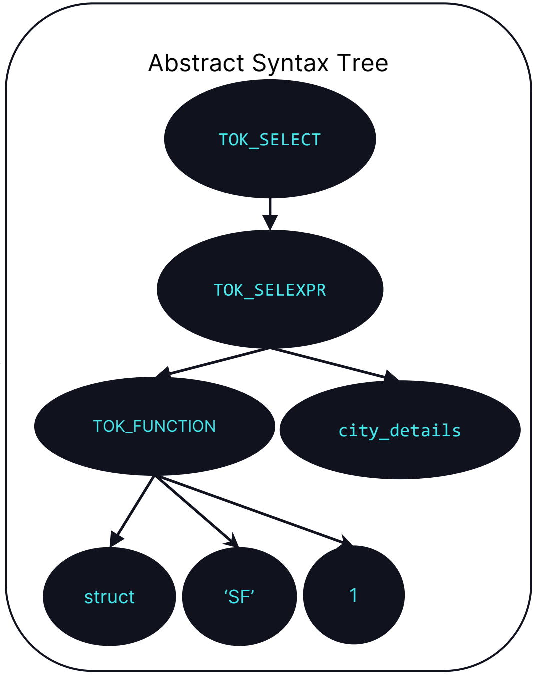
```
SET spark.sql.autoBroadcastJoinThreshold=128000000;

SELECT
    max(named_struct("col1", trip_count, "col2", city_id)).col2 as city_id
FROM
    trip_info;
```

Hive to SparkSQL Translation

```
select
struct('SF', 1)
as city_details
```

Input Hive Query



```
{
  "selectList": {
    "list": [
      ROW("SF", "1") AS
city_details
    ]
  }
}
```

SQLNode

```
{
  "selectList": {
    "list": [
      NAMED_STRUCT("col1","SF","col2", "1")
      AS city_details
    ]
  }
}
```

SparkSql compatible SQLNode

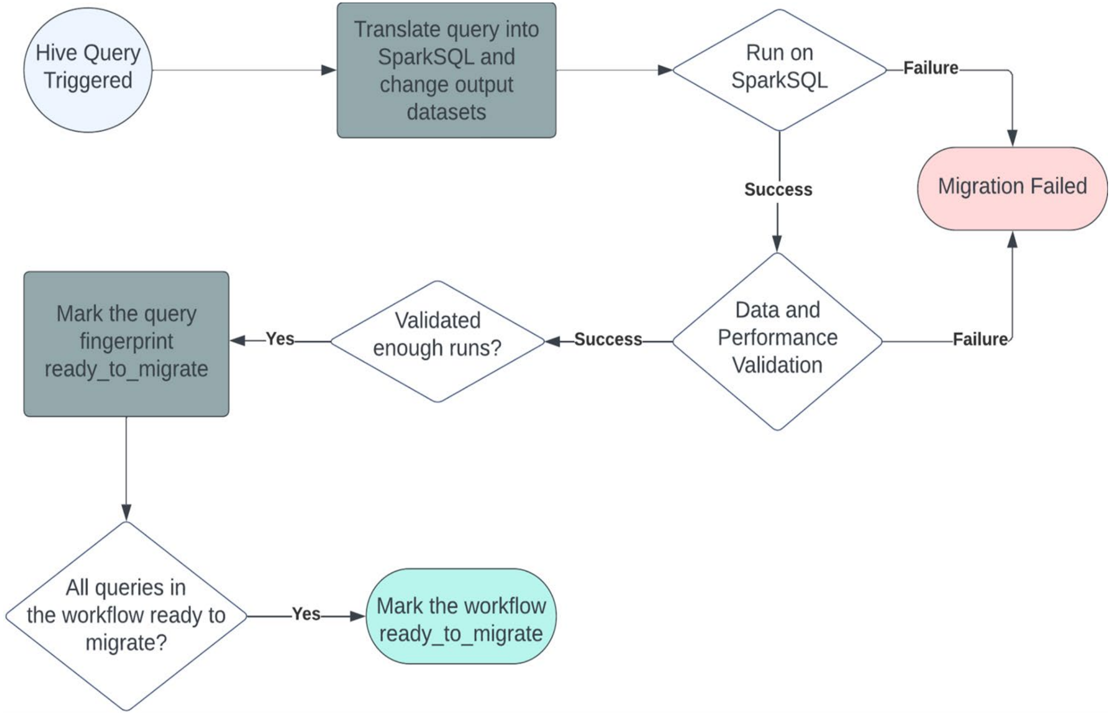
```
SELECT
 named_struct('col1', 'SF', 'col2', 1)
as city_details;
```

Output SparkSQL query

Shadow Testing

Framework

- Interception: HiveQL payload interception
- Safe Translation: Replace output/input datasets
- Execution: Serialized execution of Spark queries as per the original DAG
- Validation: Data and performance validation against production HiveQL execution


Hive Query Triggered
Translate query into SparkSQL and change output datasets
Run on SparkSQL
Failure
Migration Failed
Success
Data and Performance Validation
Failure
Success
Validated enough runs?
Yes
Mark the query fingerprint ready_to_migrate
All queries in the workflow ready to migrate?
Yes
Mark the workflow ready_to_migrate

Shadow Testing

Safe Translation

```
CREATE TABLE IF NOT EXISTS db.table_1 LIKE
db.prod_table;


INSERT OVERWRITE TABLE db.table_1
SELECT * FROM db.prod_table;


INSERT OVERWRITE TABLE db.table_2
SELECT * FROM db.table_1;
```

```
CREATE TABLE IF NOT EXISTS
migration.db_table_1 LIKE db.prod_table;


INSERT OVERWRITE TABLE migration.db_table_1
SELECT * FROM db.prod_table;

CREATE TABLE IF NOT EXISTS migration.table_2
LIKE db.table_2;

INSERT OVERWRITE TABLE migration.db_table_2
SELECT * FROM migration.db_table_1
```

Shadow Testing

Limitations

- Race conditions (Table or view not found)

```
CREATE EXTERNAL TABLE source_table
LOCATION("hdfs://external-location");

INSERT OVERWRITE dest_table
SELECT * FROM source_table;

DROP TABLE source_table;
```

Shadow execution fails with
"Table or view not found"
when it tries to read from
source_table!

Shadow Testing

Challenges

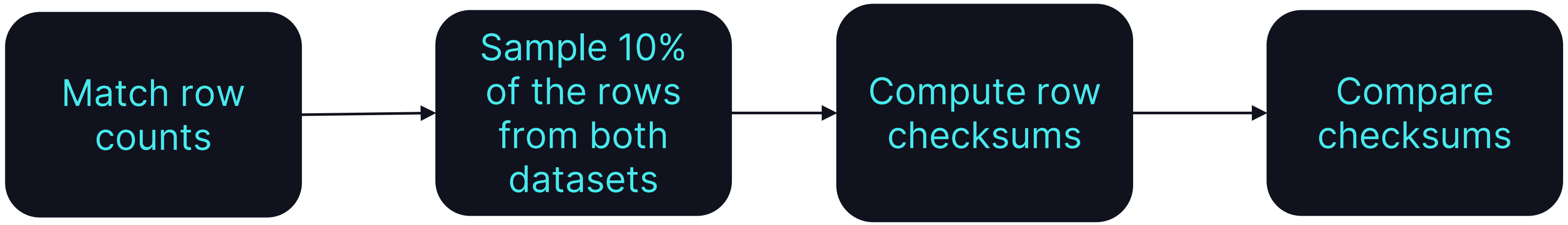
- Race conditions (Table or view not found)
  - Requires multiple runs via Shadow testing framework
- Load data queries
  - File is moved from original location to table/partition location, shadow execution fails with *FileNotFoundException*
  - **Solution:** Copied file to a temporary location in Hive for all load data queries

Shadow Testing

Challenges

- Race conditions (Table or view not found)
    - Requires multiple runs via Shadow testing framework
- Load data queries
    - File is moved from original location to table/partition location, shadow execution fails with *FileNotFoundException*
    - **Solution:** Copied file to a temporary location in Hive for all load data queries
- Schema mismatch
    - DDL (CREATE TABLE IF NOT EXISTS) in the source code not uptodate
    - Shadow dataset is created with an outdated schema
    - **Solution:** Identify such tables from failure logs and configure the correct schema in shadow testing framework

Data Validation



Data Validation

Challenges

- Floating point arithmetics
  - Round to certain precision before taking checksum
  - Add tolerance of 1% in the tests
  - Threshold on maximum difference in a columns across all rows

Data Validation

Challenges

- Floating point arithmetics
  - Round to certain precision before taking checksum
  - Add tolerance of 1% in the tests
  - Threshold on maximum difference in a columns across all rows
- Stringified JSON
  - Custom UDF to create ordered json with precision loss while sampling rows

Data Validation

Challenges

- Floating point arithmetics
    - Round to certain precision before taking checksum
    - Add tolerance of 1% in the tests
    - Threshold on maximum difference in a columns across all rows
- Stringified JSON
    - Custom UDF to create ordered json with precision loss while sampling rows
- Non-deterministic functions like row_number, rand, collect_list, collect_set, current_timestamp
    - Identify and exclude the columns from data validation

Data Validation

Challenges

- Floating point arithmetics
    - Round to certain precision before taking checksum
    - Add tolerance of 1% in the tests
    - Threshold on maximum difference in a columns across all rows
- Stringified JSON
    - Custom UDF to create ordered json with precision loss while sampling rows
- Non-deterministic functions like row_number, rand, collect_list, collect_set, current_timestamp
    - Identify and exclude the columns from data validation
- Frequently updated datasets/Circular dependencies
    - Snapshot input datasets, run both Hive and SparkSQL payloads on snapshotted datasets

Bridging the Gap Between Hive & Spark

Hive-Spark Disparity

Execution Failures

Problems	Solutions
DDLs unsupported in SparkSQL: ● ALTER TABLE DROP PARTITION(datestr< '2024-04-01') ● ALTER TABLE CLUSTERED BY	HiveDriver support in SparkSQLRunner
Hive Built-in functions' explicit registration required in Spark	Implicit Hive built-in discovery and registration added in Spark. Discovery order: Spark Built-in > Hive Built-in > UDFs
Group by on non-orderable data types (maps/structs)	Ported SPARK-34819
Out of memory errors	● Reduce target partition size ● Retry with increased executor memory

Hive-Spark Disparity

Data validation Failures

Problem	Hive	Spark	Solution

Boolean <> String Conversion	true => “TRUE”
	false => “FALSE”
	“” => false
	“any_string” => true
		true => “true”
		false => “false”
		“false” => false
		“true” => true
			Introduced behaviour on par with Hive in Spark backed by a config

Timestamp <> BigInt/Double Conversion/Coercion	cast(1714542982 as timestamp) => 1970-01-20 20:15:42.982	cast(1714542982 as timestamp) => 2024-05-01 11:26:22	Introduced behaviour on par with Hive in Spark backed by a config
Partition Schema vs Parquet Schema Preference	Partition schema > Parquet Schema	Parquet schema > Partition Schema	Identified and excluded these columns from data validation.
Skip header in CSV tables	Respects “skip.header.line.count“ in table properties	-	Modified HadoopTableReader to respect “skip.header.line.count” while creating RDD
Difference in behaviour of built-in functions	regexp_like('x', null) => false	regexp_like('x', null) =>null	Replace with Hive built-in in translation

Hive-Spark Disparity

Performance Gotchas

- BroadcastNestedLoopJoin
  - NOT IN vs NOT EXISTS
  - Solution: Required query rewrites
- Merge ORC files
  - Merging ORC files is a metadata operation in Hive and hence very efficient
  - Not solved as ORC is deprecated at Uber in favor of Parquet
- Stats Autogather
  - Spark doesn’t compute and populate stats usable by Hive
  - Downstream Hive workflows are degraded because of non-availability of stats
  - **Solution:** Started computing and updating Hive usable stats in Spark

Hive-Spark Disparity

Handling small output files

Problem

- Too many small files created by Spark causing namespace quota issues & increased HDFS directory listing latency
- Hive runs a conditional stage to merge files based on the following configs:
  - SET hive.merge.sparkfiles = true
  - SET hive.merge.smallfiles.avgsize = 128000000
  - SET hive.merge.size.per.task = 128000000

Hive-Spark Disparity

Handling small output files

Solution

- Added Rebalance in the logical plan of SparkSQL write queries by default before .
- Wrapped DataWritingCommand's child plan with Rebalance

.

Original Plan

```
CreateDataSourceTableAsSelectCommand `default`.`table1`, ErrorIfExists, [col1, col2]
+- Project [col1#20, col2#21]
  +- SubqueryAlias spark_catalog.default.tmp
    +- Relation default.tmp[col1#20,col2#21] parquet
```

Modified Plan

```
CreateDataSourceTableAsSelectCommand `default`.`table1`, ErrorIfExists, [col1, col2]
+- RebalancePartitions 200, false
  +- Project [col1#20, col2#21]
    +- SubqueryAlias spark_catalog.default.tmp
      +- Relation default.tmp[col1#20,col2#21] parquet
```

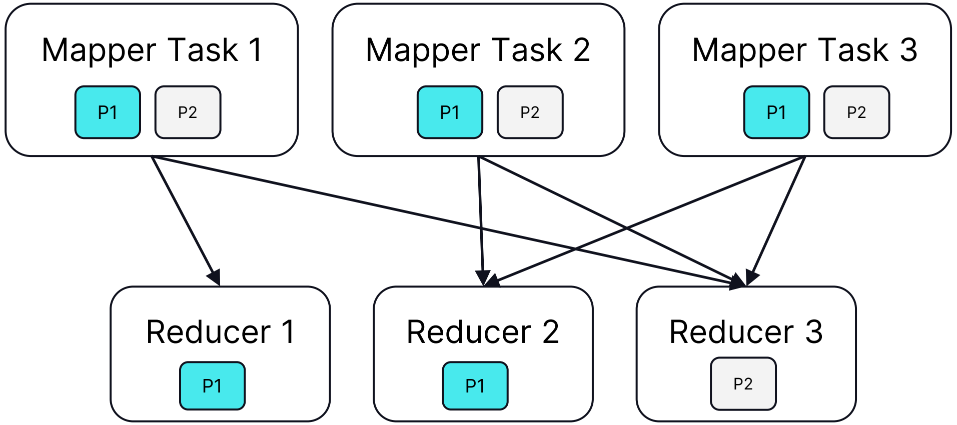
Hive-Spark Disparity

Handling small output files

Solution

- Added Rebalance in the logical plan of SparkSQL write queries by default before .
- Wrapped DataWritingCommand's child plan with Rebalance
- Tweaked AQE rules *CoalesceShufflePartitions* and *OptimizeSkewInRebalancePartitions* to coalesce/split partitions based on the file target size for rebalance instead of *AdvisoryPartitionSizeInBytes*.

.



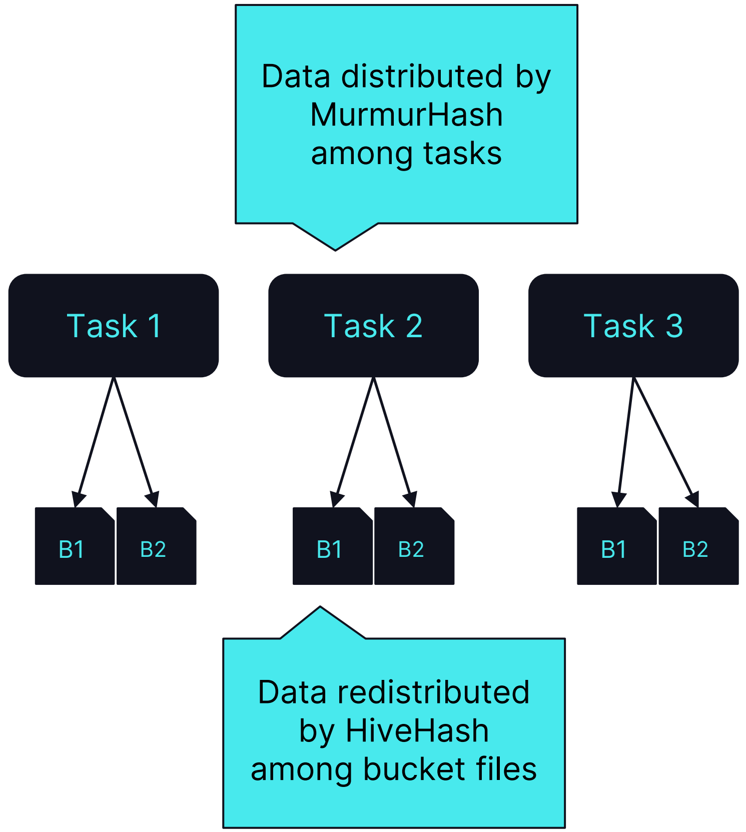
Skewed partition P1 split by OptimizeSkewInRebalancePartitions

Hive-Spark Disparity

Bucketed Tables

Problem

- Hive and Spark use different hashing algorithms for bucketing
- Spark supports writing both *HiveHash* and *MurmurHash* while writing to bucketed tables but ends up creating too many files (#tasks * #buckets) while using *HiveHash*
- Presto doesn't recognize Spark buckets


Data distributed by MurmurHash among tasks
Task 1
Task 2
Task 3
B1
B2
B1
B2
B1
B2
Data redistributed by HiveHash among bucket files

Hive-Spark Disparity

Bucketed Tables

Problem

- Hive and Spark use different hashing algorithms for bucketing
- Spark supports writing both *HiveHash* and *MurmurHash* while writing to bucketed tables but ends up creating too many files (#tasks * #buckets) while using *HiveHash*
- Presto doesn't recognize Spark buckets

Solution

- Decided to stick to *HiveHash* for bucketed tables as majority read use cases were in Presto.
- Added support for *HiveHash* in Rebalance to reduce the number of files.
- TBD: Extend *HiveHash* support to all shuffle stages for bucket table reads.

Results

100%	80%	4M	50%
Interactive Workloads Migrated	ETL Workflows Migrated	Monthly Queries Migrated	Reduction in Runtime & Cost

Future Work

- JDBC/ODBC access for SparkSQL
- Fast fail on semantic issues for SQL in BI tools
- Optimize compression while shuffling data before write
- Atomic updates to prevent *FileNotFoundException* in case of concurrent write and reads

Thank You!