Robust and Scalable ETL over Cloud Storage

Eric Liang
Databricks

What is ETL?

- The most common Spark use case

Simple ETL example

```
spark.read.csv("s3://source")
    .groupBy(...)
  .agg(...)
    .write.mode("append")
    .parquet("s3://destination")
```

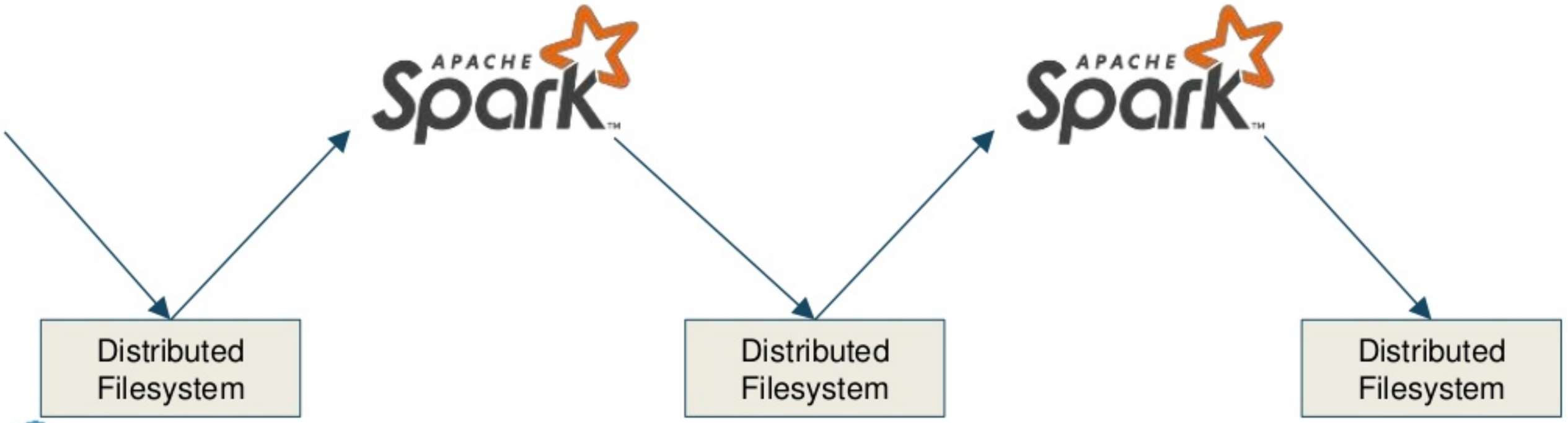
Extract

Transform

Load

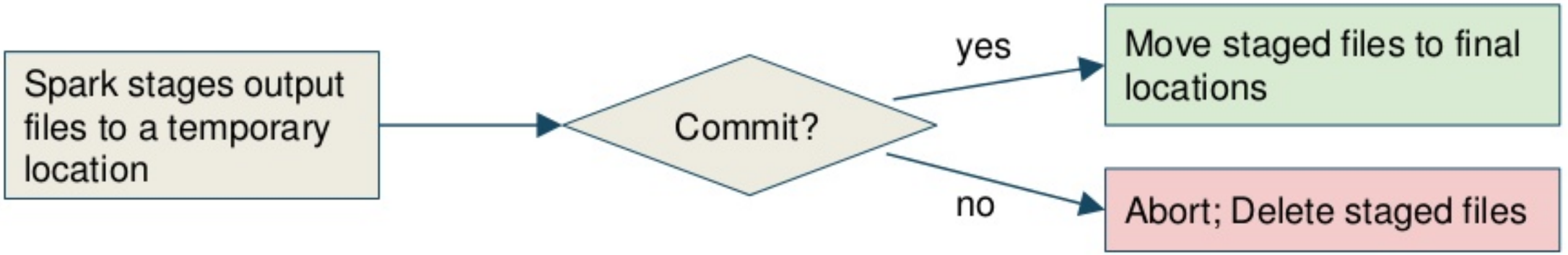
ETL jobs are commonly chained

- Output of one job may be the input of another



Reliable output is critical

- With chained ETL jobs, output should be all-or-nothing, i.e. committed atomically
- Otherwise, failure corrupts downstream jobs



Atomic commit with 

- An important part of the commit protocol is actually moving the files
- HDFS supports atomic metadata operations
  - allows a staged file / directory to be moved final location in one metadata operation
- Spark's `HadoopMapReduceCommitProtocol` uses series of files moves for Job commit
  - practically all-or-nothing when using HDFS

What about the Cloud?

Reliable output works on HDFS, but what about the Cloud?

- Option 1: run HDFS worker on each node (e.g. EC2 instance)
  - –replicate on-prem Spark deployment in Cloud
- Option 2: Use Cloud-native storage (e.g. S3)
  - –S3 / GCS / Azure blob store are not filesystems
  - –Closer to key-value stores

Object stores as Filesystems

- Not so hard to provide Filesystem API over object stores such as S3
  - –e.g. S3A Filesystem
- Traditional Hadoop applications / Spark continue to work over Cloud storage using these adapters
- What do you give up (reliability, performance)?

The remainder of this talk

1. Why Cloud-native storage is preferable over HDFS
2. The performance and reliability challenges when using Cloud storage for ETL
3. What we're doing about it at Databricks

Evaluating storage systems

1. Total cost of ownership
2. SLA (Availability and durability)
3. Performance
4. Consistency

Let's compare HDFS and S3

(1) Total cost of ownership

- Storage cost + human cost
  - S3 storage is ~4x cheaper than HDFS on EBS
  - S3 is also fully managed, in contrast to HDFS which requires Hadoop engineers or vendor support
- S3 also fully elastic
- Overall S3 is likely at least 10X cheaper

(2) Availability and Durability

- Amazon claims 99.999999999% durability, and 99.99% availability.
  - Unlikely to achieve this running your own HDFS cluster without considerable operational expertise

(3) Performance

- Data plane throughput
  - HDFS offers higher per-node throughput w/locality
  - S3 throughput scales to needs => better price:perf
- Control plane / metadata throughput
  - S3: Listing files much slower
  - S3: Renaming files requires copy and is not atomic
    - renames get slower with the size of the file
    - increases window of failure during commit

(4) Consistency

- HDFS provides strong consistency (reads guaranteed to reflect previous writes)
- S3 offers read-after-write for some operations, eventual consistency for others

Cloud storage is preferred

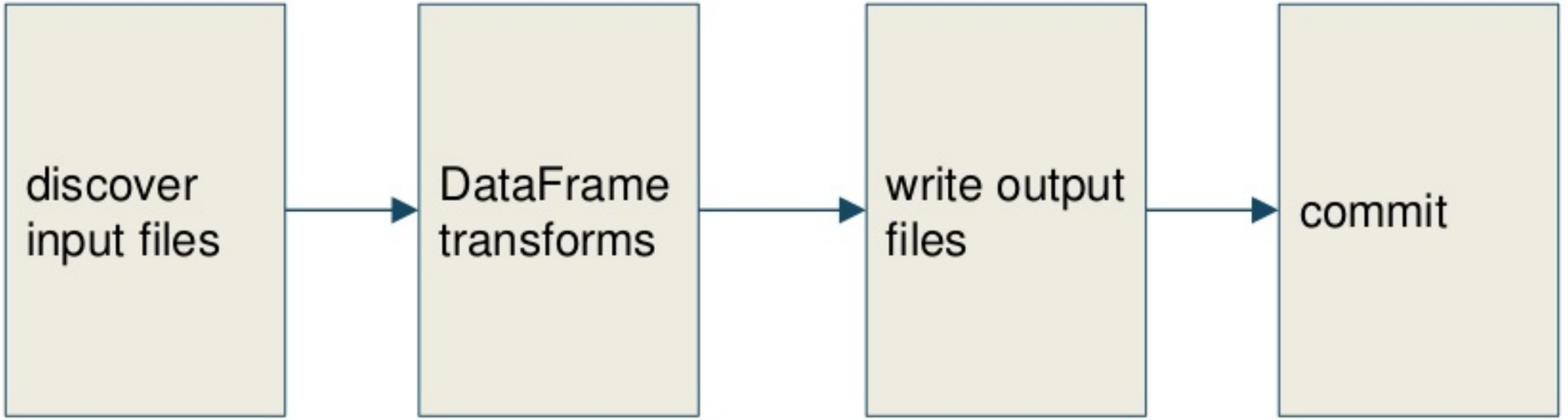
- Cloud-native storage wins in cost and SLA
  - better price-performance ratio
  - more reliable
- However it brings challenges for ETL
  - lower metadata performance
  - lack of atomic operations
  - eventual consistency

ETL job example

- To make these issues concrete, let's look at an example:

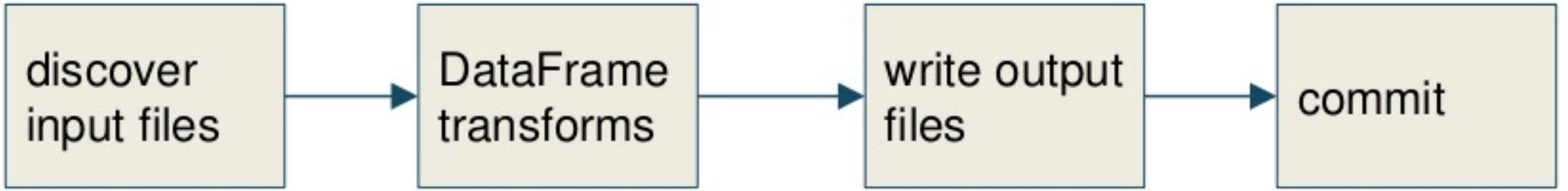
```
val inputDf = spark.table("hourly_metrics")
inputDf.filter("date = '2017-02-09'")
    ...
    .write
    .mode("overwrite")
    .parquet("s3://daily_summary")
```

How the job is executed



- Cloud storage issues can affect each phase

Cloud ETL issues that arise



Reliability:

i) some input files may not be found due to eventual consistency

ii) during the job run, (external) readers may observe missing or partially written output files

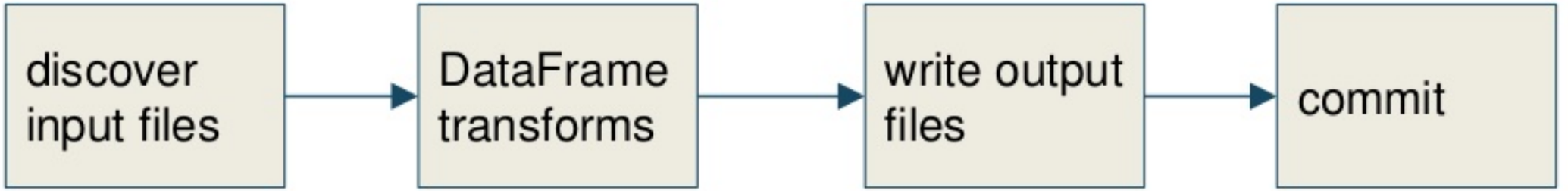
iii) must move staged files to their final location: slow on cloud storage. Workarounds such as DirectOutputCommitter leave partial output around on failures.

Performance:

i) finding all the input files takes a long time since list calls are slow

ii) not safe to enable Spark speculation against S3, so stragglers slow down job

Cloud ETL issues that arise



Reliability:

i) some input files may not be found due to eventual consistency

Eventual Consistency

ii) during the job run, (external) readers may observe missing or partially written output files

iii) must move staged files to their final location: slow on cloud storage. Workarounds such as DirectOutputCommitter leave partial output around on failures.

Performance:

i) finding all the input files takes a long time since list calls are slow

Metadata Performance

ii) not safe to enable Spark speculation against S3, so stragglers slow down job

Lack of atomic commit

Addressing Cloud ETL issues

Can we avoid tradeoffs to using Cloud storage?
Yes! leverage external services to provide additional guarantees

1. Eventual consistency
2. Metadata performance
3. Lack of atomic commit

Eventual Consistency

- Problem for many cloud storage systems

	S3	Azure Blob Store	Google Cloud Storage
Read-after-write consistent for single object	New objects only	Yes	Yes
Read-after-write consistent for LIST after PUT	No	No	No

Eventual Consistency

```
> spark.range(100)
      .write
      .mode("overwrite")
      .parquet("s3://test")
> print spark.read.parquet("s3://test").count()
```

Strongly consistent: always prints 100

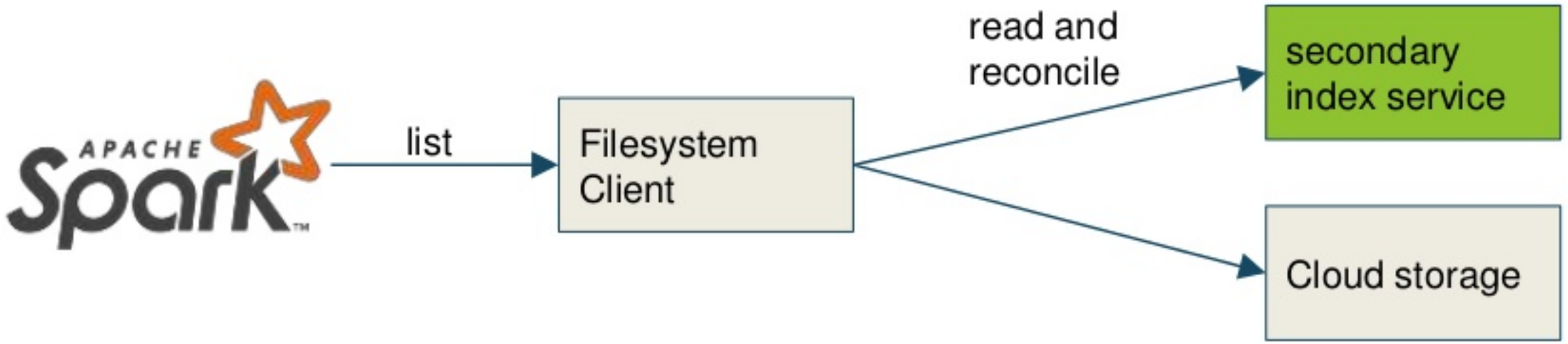
Eventually consistent: can print

<100 (no LIST-after-PUT consistency)

>100 if there was previous data (no LIST-after-DELETE)

Eventual Consistency

- Open source solutions: S3mper, S3Guard (dev)
- Vendor implementations: EMRFS, DBFS
- All use a strongly consistent secondary index



Addressing Cloud ETL issues

1. Eventual consistency
2. Metadata performance
3. Lack of atomic commit

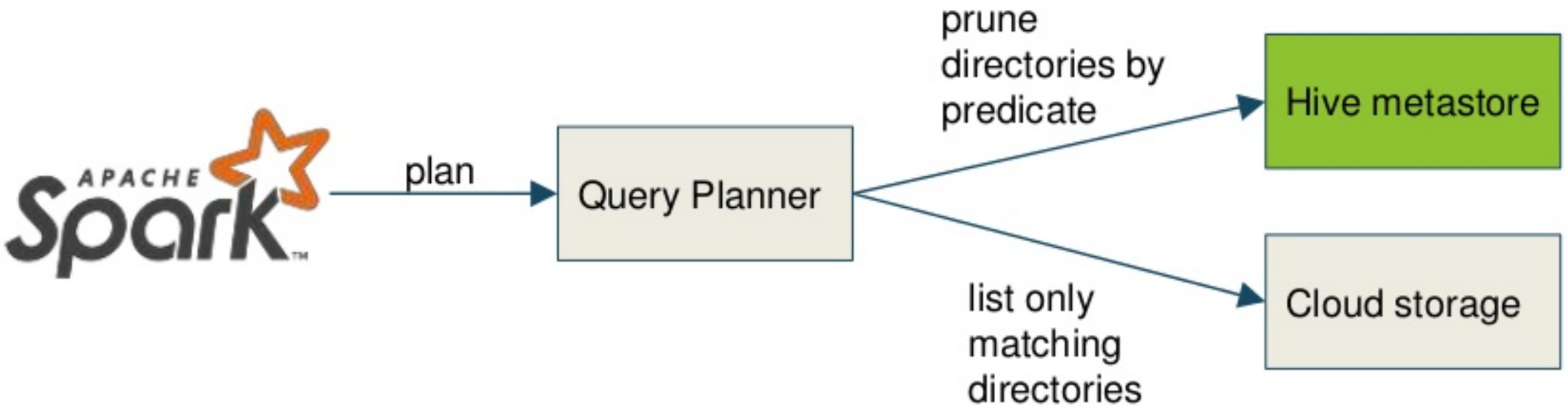
Metadata performance

```
> spark.read.table("hourly_metrics")
      .filter("date = '2017-02-09'")
      .count()
```

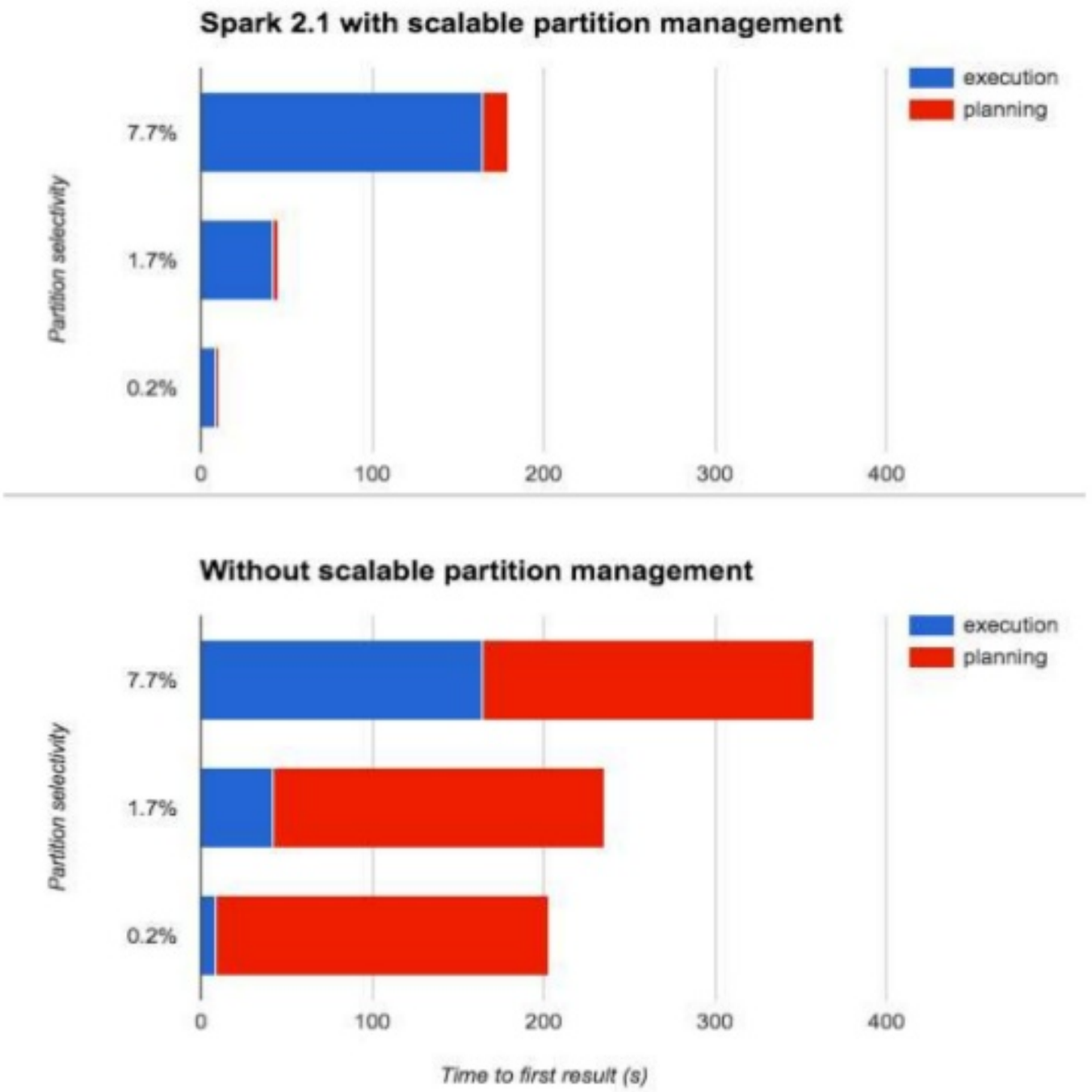
- When using cloud storage, planning phase where Spark lists files may be very slow
- Filter on partition field doesn't help since it is applied after the list of files is computed

Metadata performance

- Scalable partition handling in Spark 2.1
- Leverage metadata catalog (e.g. Hive metastore) to avoid expensive S3 lists when possible



databricks.com/blog/2016/12/15/scalable-partition-handling-for-cloud-native-architecture-in-apache-spark-2-1.html



Metadata performance

- Beyond read performance, slowness of renames is also an issue
- Previous solution: *`DirectOutputCommitter`*
  - don't bother staging output files in temporary location
  - just write them directly to the destination
- Trades reliable output for performance
  - Removed in Spark 2.0 due to these issues

Addressing Cloud ETL issues

1. Eventual consistency
2. Metadata performance
3. Lack of atomic commit

Atomic commit

```
> spark.range(100)
      .write
      .mode("overwrite")
      .parquet("s3://test
")
```

```
> print spark.read
    .parquet("s3://test")
    .count()
```

Atomic: always prints prev value -or- 100

Non-atomic: can print prev value, zero, 100, -or-
any value in between (partially committed writes)

Atomic commit

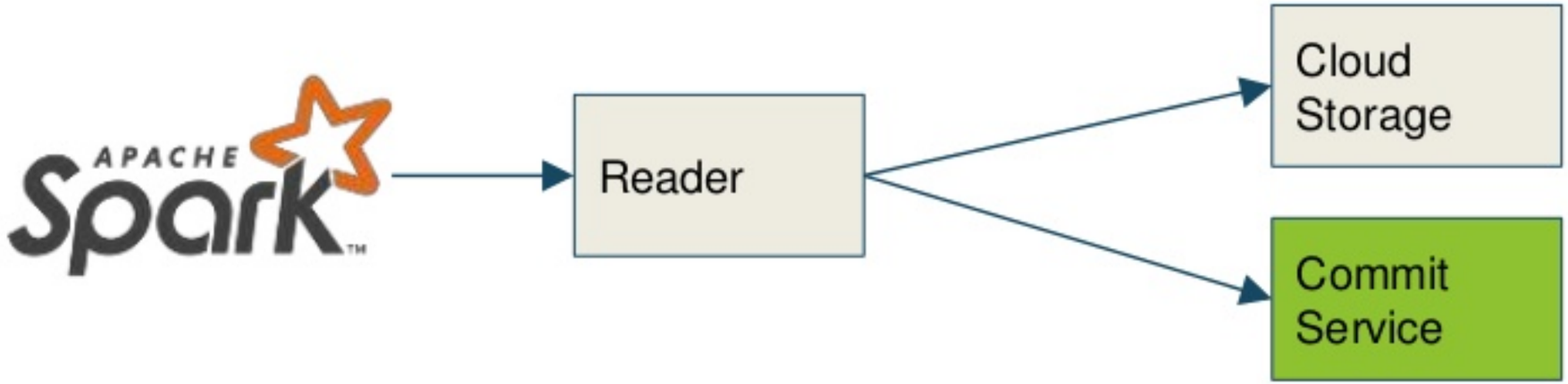
- Want readers to see only committed data
  - atomic (all-or-nothing) append / overwrites
- Failures should not affect output data
- Also nice to have
  - high performance
  - safe task speculation

Databricks Commit Service

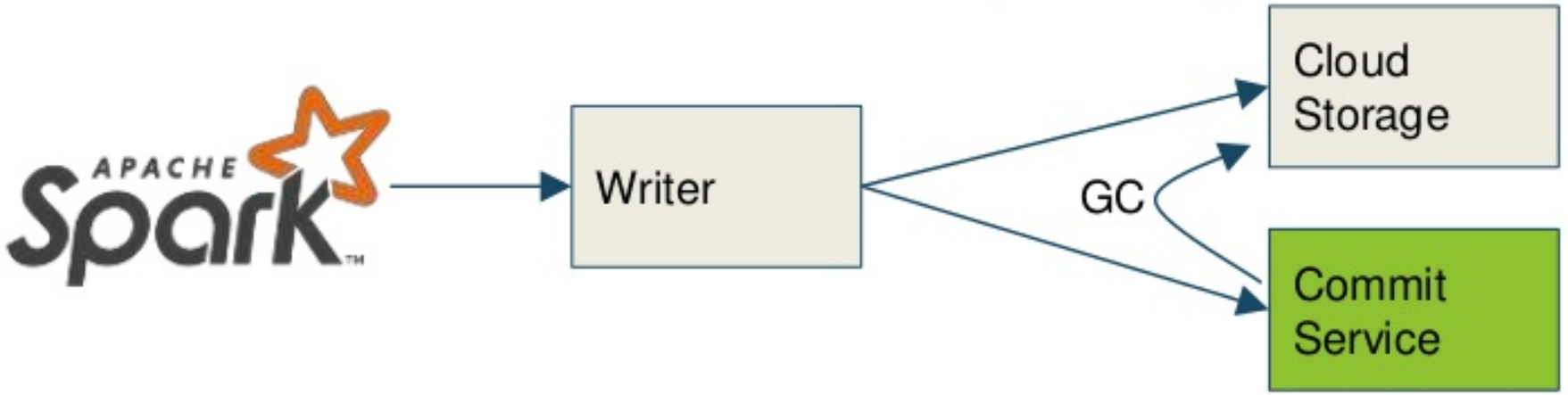
- Provides both high performance and robust output
- Basic idea: track the state of output files explicitly
  - Output files atomically visible on commit
  - Allow files to be written out in-place
- Fully backwards compatible
  - directory-level markers for compat with Hive DDL

Databricks Commit Service

Reader consults with commit service to filter out uncommitted files



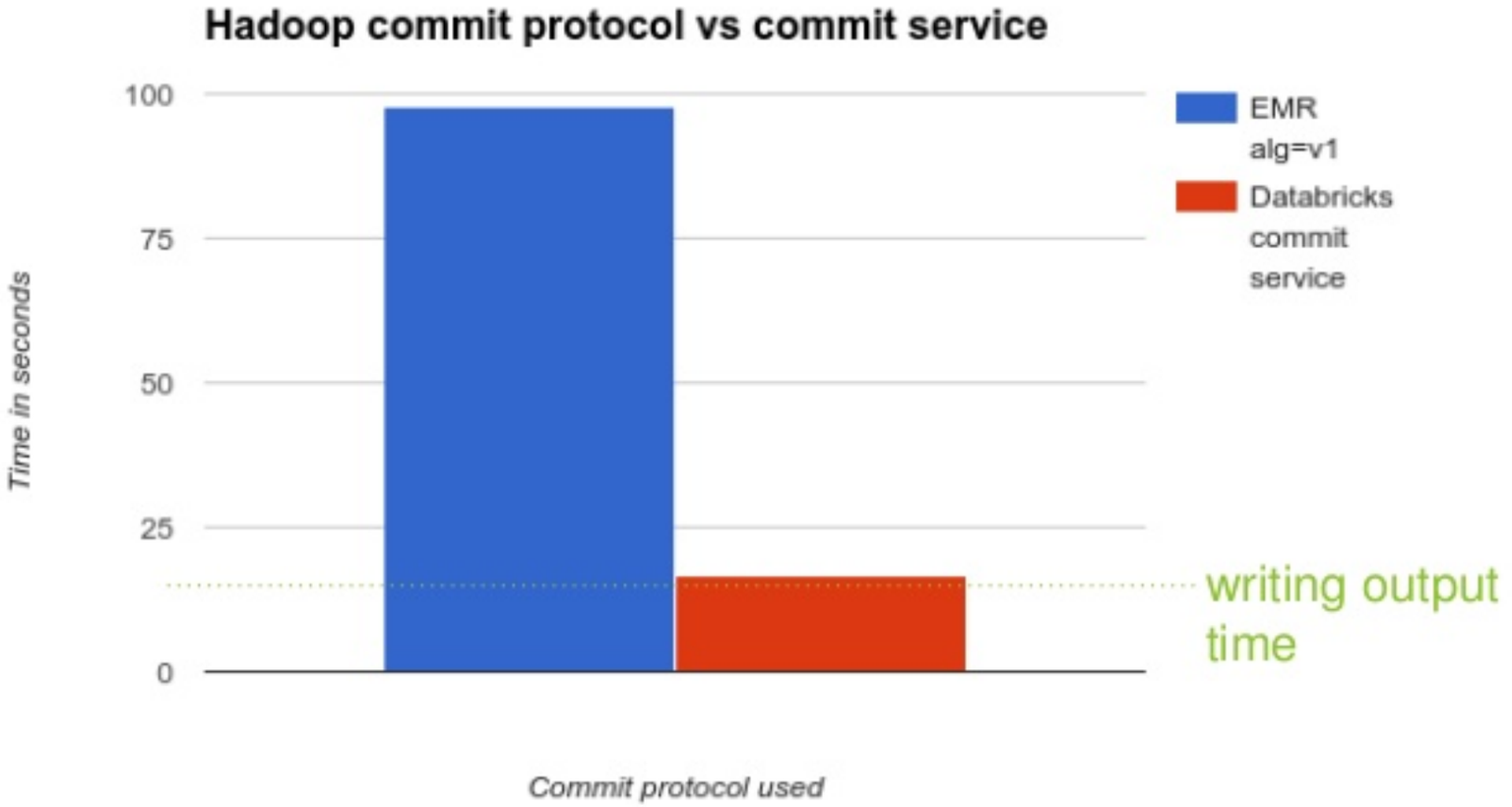
Writer informs commit service of file creations, deletions, and commit



Performance without tradeoffs

Commit Service provides *both* high performance and reliable (atomic) commit

Benchmark here: writing 1000 files in a Spark job



Summary

- There are ETL issues when using cloud storage
- But you can get both consistency and atomicity without sacrificing performance
- Databricks commit service in preview, integration will be enabled by default soon for Spark 2.1

Thank You.

Try Spark 2.1 on Community Edition:
databricks.com/ce