Transactional Writes to Cloud Storage

Eric Liang

6/7/2017

databricks

About Databricks

TEAM

Started Spark project (now Apache Spark) at UC Berkeley in 2009

MISSION

Making Big Data Simple

PRODUCT

Unified Analytics Platform

Overview

Deep dive on challenges in writing to Cloud storage (e.g. S3) with Spark

- Why transactional writes are important
- Cloud storage vs HDFS
- Current solutions are tuned for HDFS
- What we are doing about this

What are the requirements for writers?

Spark: unified analytics stack

Common end-to-end use case we've seen: chaining Spark jobs

ETL => analytics

ETL => ETL

Requirements for writes come from downstream Spark jobs

Writer example (ETL)

```
spark.read.csv("s3://source")
```
Extract

```
    .groupBy(...)
    .agg(...)
```
Transform

```
    .write.mode("append")
    .parquet("s3://dest")
```
Load

Reader example (analytics)

```
spark.read.parquet("s3://dest")
    .groupBy(...)
    .agg(...)
    .show()
```

Extract

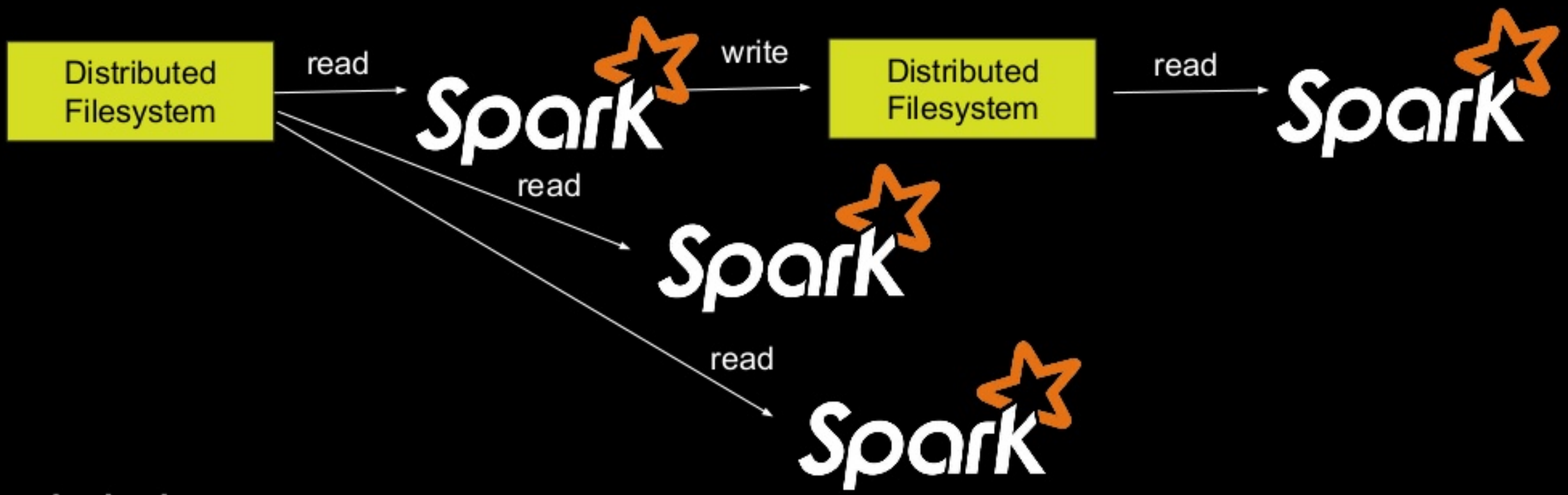
Transform

Reader requirement #1

- With chained jobs,
- Failures of upstream jobs shouldn't corrupt data

Reader requirement #2

- Don't want to read outputs of in-progress jobs



Reader requirement #3

- Data shows up (eventual consistency)

FileNotFoundException

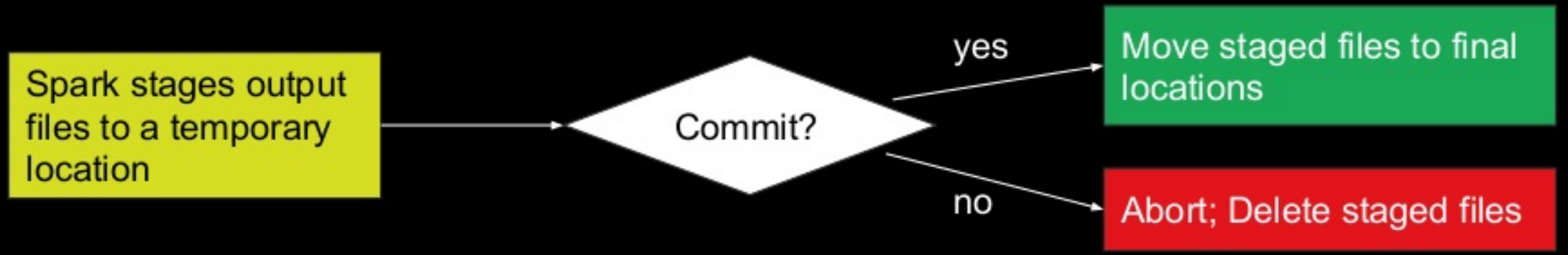
Summary of requirements

Writes should be committed **transactionally** by Spark

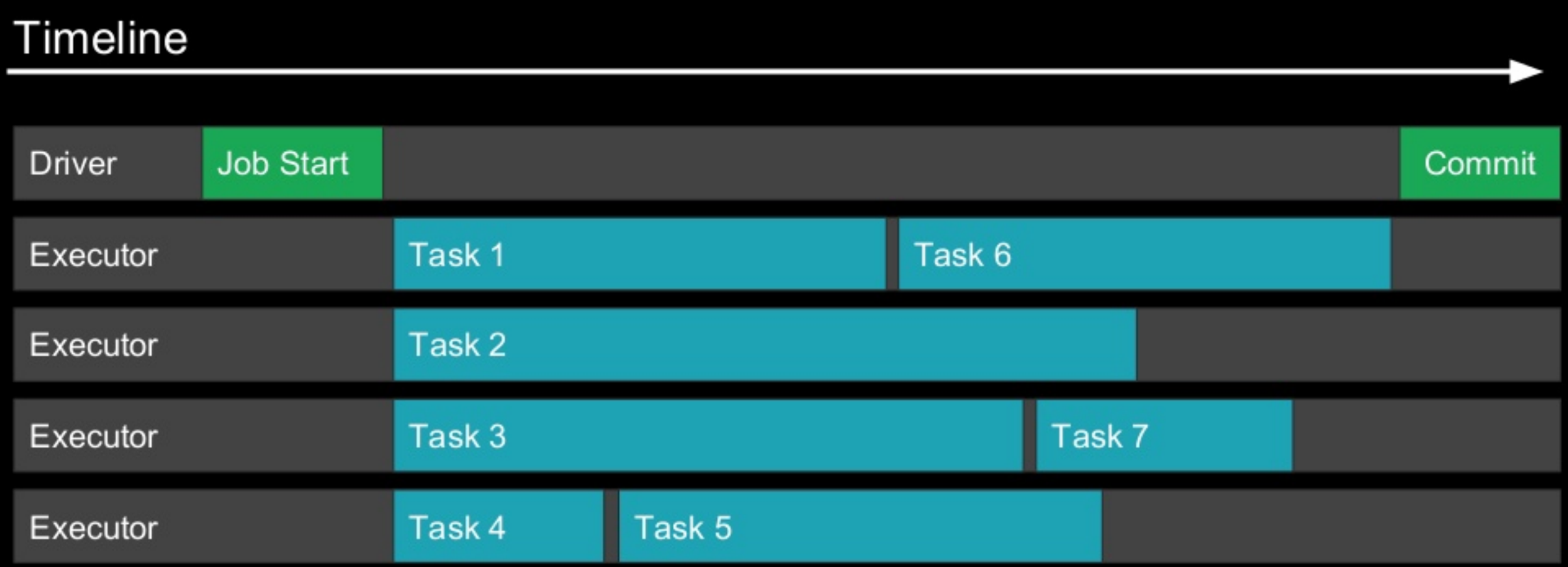
- Writes commit atomically
- Reads see consistent snapshot

How Spark supports this today

- Spark's *commit protocol* (inherited from Hadoop) ensures reliable output for jobs
- `HadoopMapReduceCommitProtocol:`



Spark commit protocol



Commit on HDFS

- HDFS is the Hadoop distributed filesystem
- Highly available, durable
- Supports fast atomic metadata operations
  - e.g. file moves
- `HadoopMapReduceCommitProtocol` uses series of files moves for Job commit
  - on HDFS, good performance
  - close to transactional in practice

Does it meet requirements?

Spark commit on HDFS:

1. No partial results on failures => yes*
2. Don't see results of in-progress jobs => yes*
3. Data shows up => yes

* window of failure during commit is small

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)

Option 2: Use Cloud-native storage (e.g. S3)

- Challenge: systems like S3 are not true filesystems

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 (transactionality, performance)?

The remainder of this talk

1. Why HDFS has no place in the Cloud

2. Tradeoffs when using existing Hadoop commit protocols with Cloud storage adapters

3. New commit protocol we built that provides transactional writes in Databricks

Evaluating storage systems

1. Cost
2. SLA (availability and durability)
3. Performance

Let's compare HDFS and S3

(1) Cost

- Storage cost per TB-month
  - HDFS: $103 with d2.8xl dense storage instances
  - S3: $23
- Human cost
  - HDFS: team of Hadoop engineers or vendor
  - S3: $0
- Elasticity
- Cloud storage is likely >10x cheaper

(2) SLA

- Amazon claims 99.99% availability and 99.999999999% durability.
- Our experience:
  - S3 only down twice in last 6 years
  - Never lost data
- Most Hadoop clusters have uptime <= 99.9%

(3) Performance

- Raw read/write performance
  - HDFS offers higher per-node throughput with disk locality
  - S3 decouples storage from compute
    - performance can scale to your needs
- Metadata performance
  - S3: Listing files much slower
    - Better w/scalable partition handling in Spark 2.1
  - S3: File moves require copies (expensive!)

Cloud storage is preferred

- Cloud-native storage wins in cost and SLA
  - better price-performance ratio
  - more reliable
- However it brings challenges for ETL
  - mostly around transactional write requirement
  - what is the right commit protocol?

https://databricks.com/blog/2017/05/31/top-5-reasons-for-choosing-s3-over-hdfs.html

Evaluating commit protocols

- Two commit protocol variations in use today
  - `HadoopMapReduceCommitProtocol V1`
    - (as described previously)
  - `HadoopMapReduceCommitProtocol V2`
- ~~`DirectOutputCommitter`~~
  - deprecated and removed from Spark
- Which one is most suitable for Cloud?

Evaluation Criteria

1. **Performance** – how fast is the protocol at committing files?

2. **Transactionality** – can a job cause partial or corrupt results to be visible to readers?

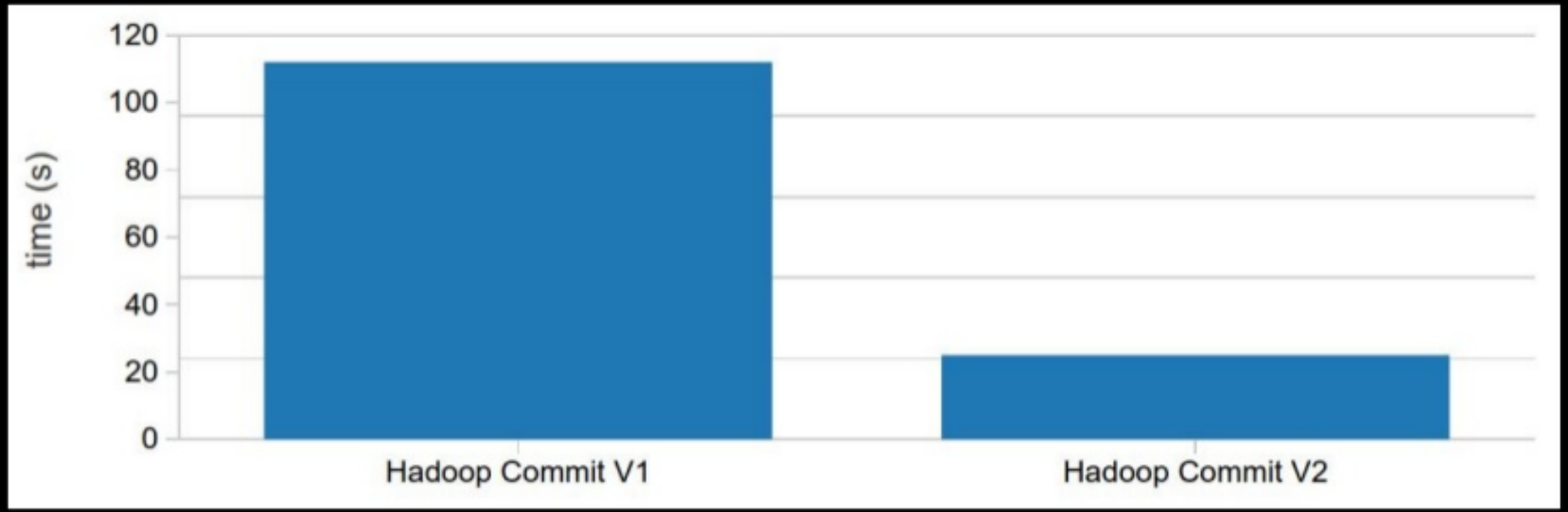
Performance Test

Command to write out 100 files to S3:

```
spark.range(10e6.toLong)
  .repartition(100).write.mode("append")
  .option(
       "mapreduce.fileoutputcommitter.algorithm.version",
       version)
  .parquet(s"s3:/tmp/test-$version")
```

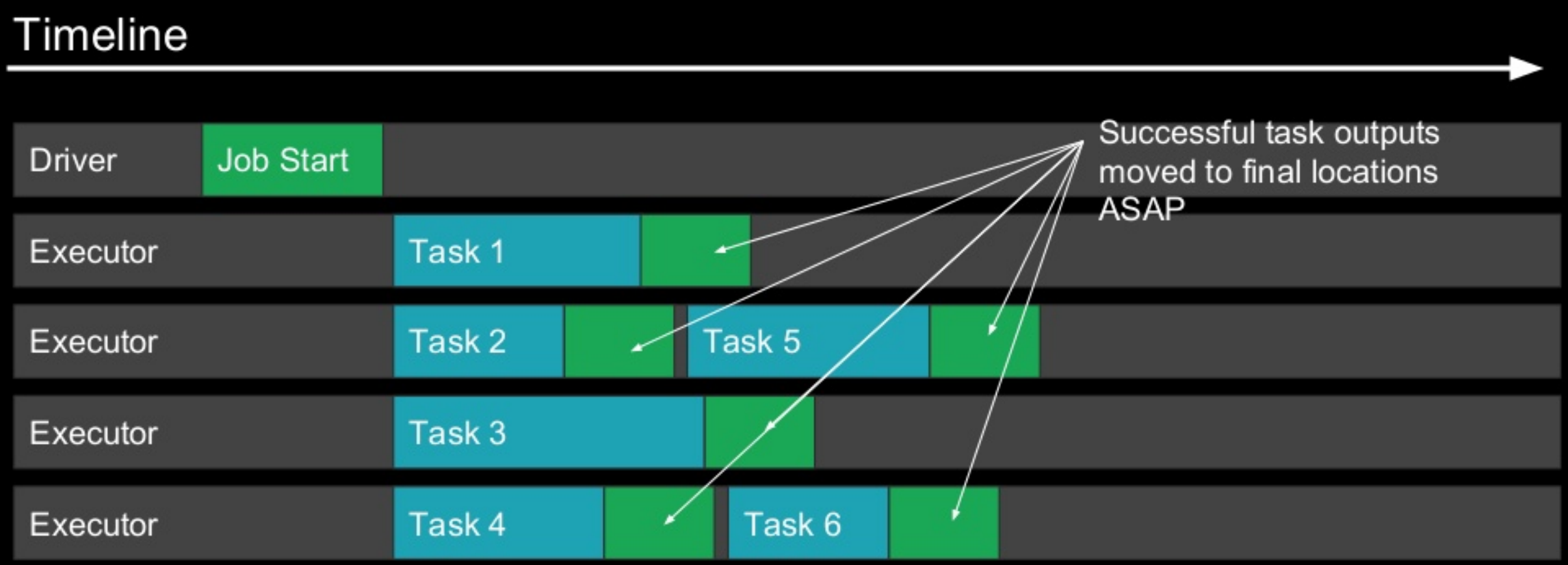
Performance Test

Time to write out 100 files to S3:

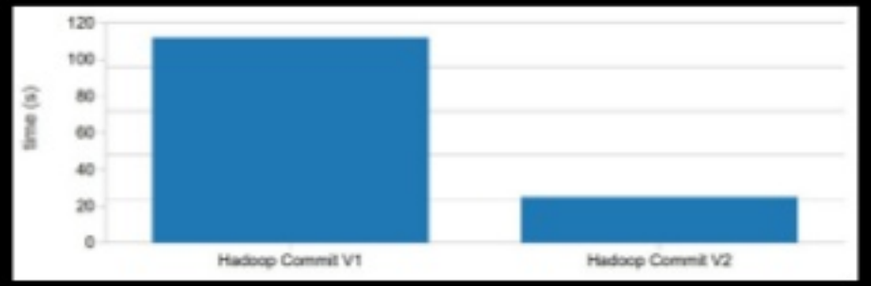


Hadoop Commit V1

Hadoop Commit V2



Performance Test



- The V1 commit protocol is slow
  - it moves each file serially on the driver
  - moves require a copy in S3
- The V2 Hadoop commit protocol is almost five times faster than V1
  - parallel move on executors

Transactionality

- Example: What happens if a job fails due to a bad task?
  - Common situation if certain input files have bad records
- The job should fail cleanly
  - No partial outputs visible to readers
- Let's evaluate Hadoop commit protocols {V1, V2} again

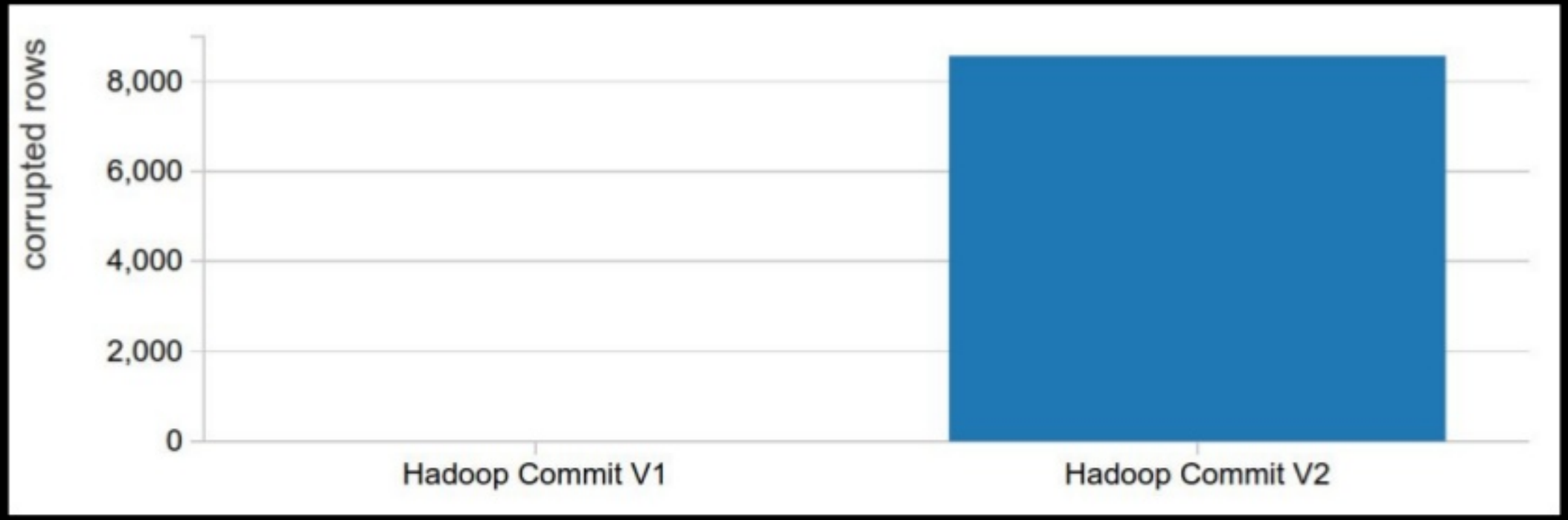
Fault Test

Simulate a job failure due to bad records:

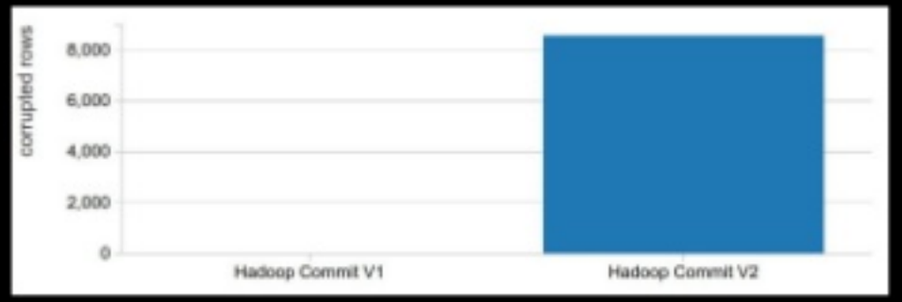
```
spark.range(10000).repartition(7).map { i =>
  if (i == 123) { throw new RuntimeException("oops!") }
  else i
}.write.option("mapreduce.fileoutputcommitter.algorithm.version
", version)
  .mode("append").parquet(s"s3:/tmp/test-$version")
```

Fault Test

Did the failed job leave behind any partial results?



Fault Test



- The V2 protocol left behind ~8k garbage rows for the failed job.
  - This is duplicate data that needs to be "manually" cleaned up before running the job again
- We see empirically that while V2 is faster, it also leaves behind partial results on job failures, breaking transactionality requirements

Concurrent Readers Test

- Wait, is V1 really transactional?

```
spark.range(1e9.toLong)
     .repartition(70)
     .write.mode("append")
     .parquet("s3:/test")
```

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

> 0
> 0
> 14285714
> 814285714
> 1000000000
```

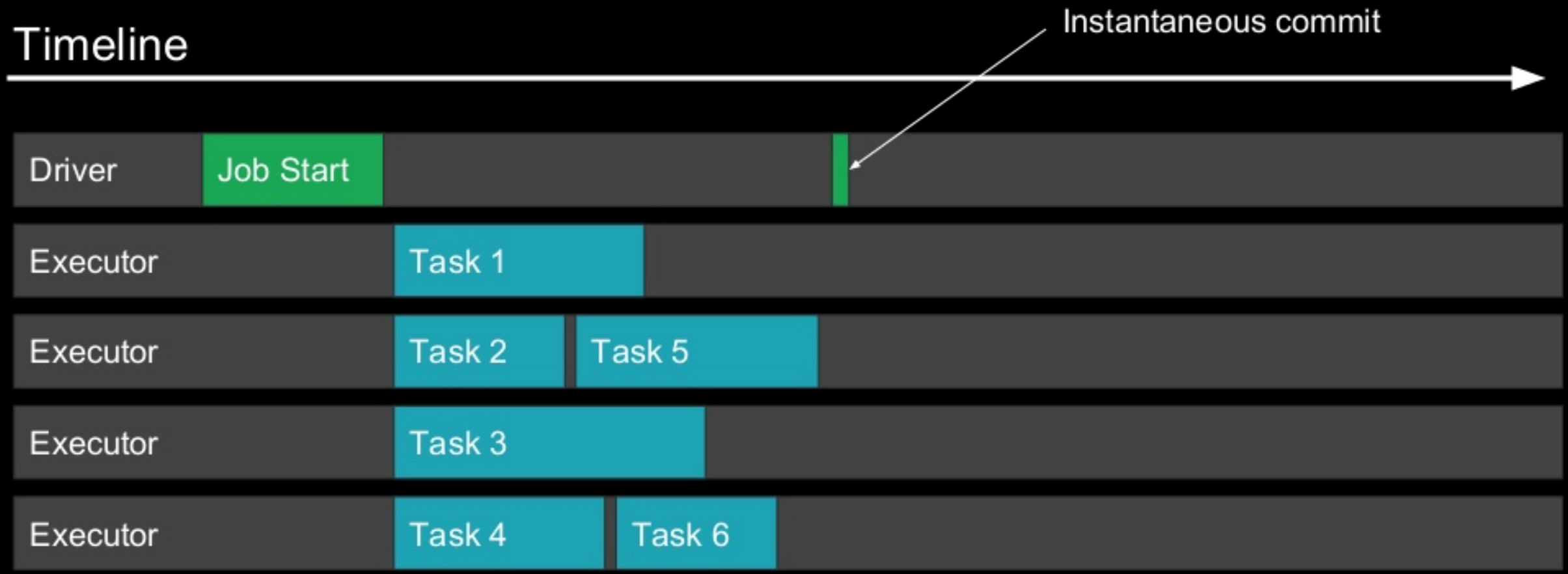
Transactionality Test

- V1 isn't very "transactional" on Cloud storage either
- Long commit phase
  - What if driver fails?
  - What if another job reads concurrently?
- On HDFS the window of vulnerability is much smaller because file moves are O(millisecond)

The problem with Hadoop commit

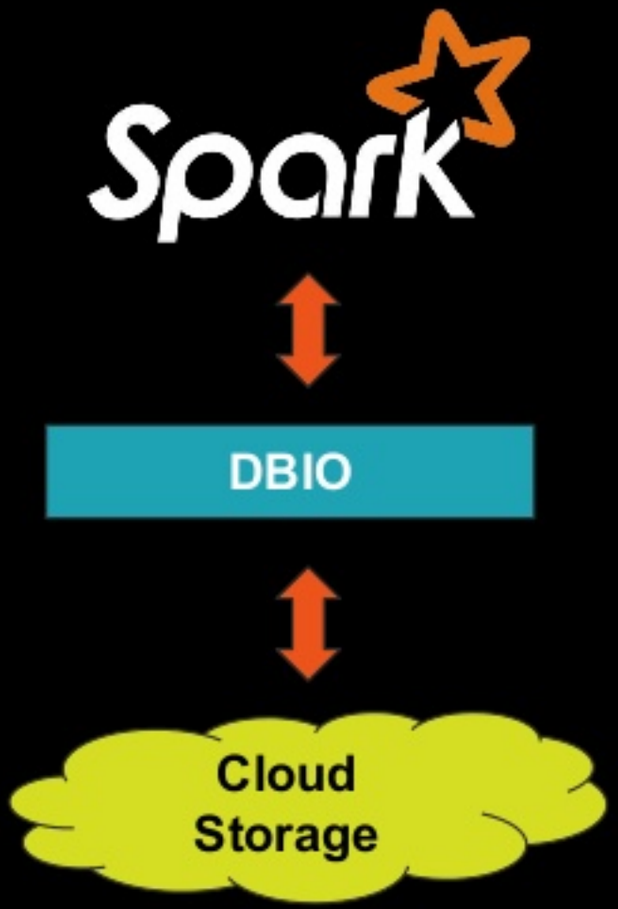
- Key design assumption of Hadoop commit V1: file moves are fast
- V2 is a workaround for Cloud storage, but sacrifices transactionality
- This tradeoff isn't fundamental
- How can we get both strong transactionality guarantees and the best performance?

Ideal commit protocol?

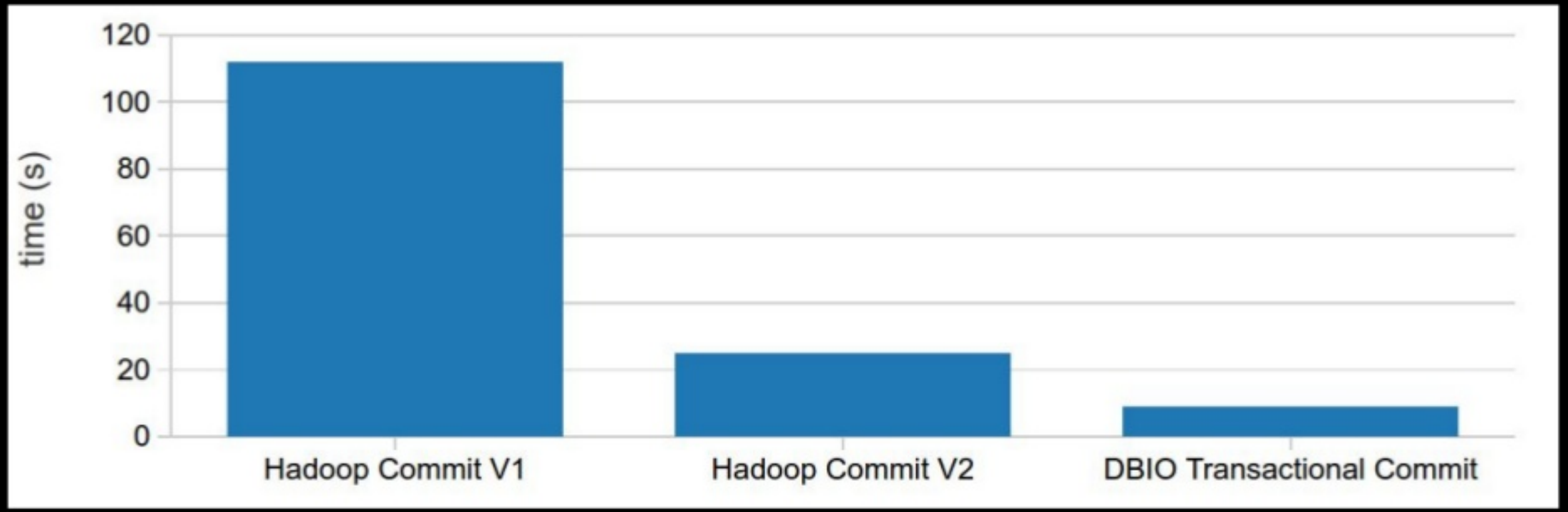


No compromises with DBIO transactional commit

- When a user writes a file in a job
  - Embed a unique txn id in the file name
  - Write the file directly to its final location
  - Mark the txn id as committed if the job commits
- When a user is listing files
  - Ignore the file if it has a txn id that is uncommitted



DBIO transactional commit



Fault analysis

	No commit protocol	Hadoop Commit V1	Hadoop Commit V2	DBIO Transactional Commit
Task failure	✗	✓	✓	✓
Job failure (e.g. persistent task failure)	✗	✓	✗	✓
Driver failure during commit	✗	✗	✗	✓

Concurrent Readers Test

- DBIO provides atomic commit

```
spark.range(1e9.toLong)
     .repartition(70)
     .write.mode("append")
     .parquet("s3:/test")
```

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

> 0
> 0
> 0
> 0
> 1000000000
```

Other benefits of DBIO commit

- High performance
- Strong correctness guarantees

+ Safe Spark task speculation

+ Atomically add and remove sets of files

+ Enhanced consistency

Implementation challenges

- Challenge: compatibility with external systems like Hive
- Solution:
  - relaxed transactionality guarantees: If you read from an external system you might read uncommitted files as before
  - %sql VACUUM command to clean up files (also auto-vacuum)
- Challenge: how to reliably track transaction status
- Solution:
  - hybrid of metadata files and service component
  - more data warehousing features in the future

Rollout status

Enabled by default starting with Spark 2.1-db5 in Databricks

Process of gradual rollout

>100 customers already using DBIO transactional commit

Summary

- Cloud Storage >> HDFS, however
- Different design required for commit protocols
- No compromises with DBIO transactional commit

See our engineering blog post for more details

https://databricks.com/blog/2017/05/31/transactional-writes-cloud-storage.html

Try Apache Spark in Databricks!

UNIFIED ANALYTICS PLATFORM

- Collaborative cloud environment
- Free version (community edition)

DATABRICKS RUNTIME 3.0

- Apache Spark - optimized for the cloud
- Caching and optimization layer - DBIO
- Enterprise security - DBES

Try for free today.
databricks.com