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

Optimizing data lakes for Apache Spark

Matthew Powers, Prognos

#UnifiedAnalytics #SparkAISummit

About

But what about those poor data scientists that work with gzipped CSV lakes 😱

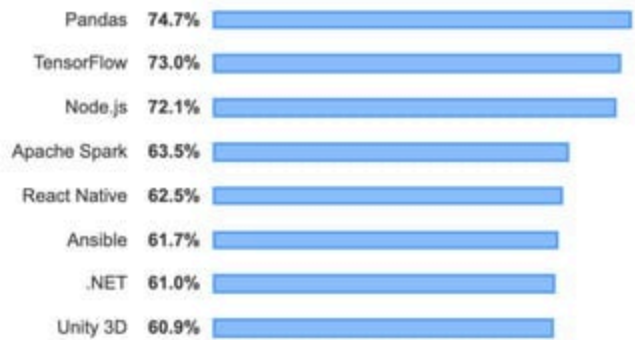
What you will get from this talk…

- Motivation to write Spark open source code
- Practical knowledge to build better data lakes

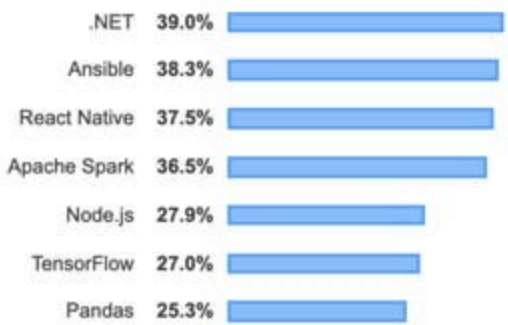
Agenda

- Community goals
- Spark open source
- Modern Scala libs
- Parquet lakes
- Incremental updates & small files
- Partitioned lakes
- Delta lakes

Loved by most



Dreaded by some



Source: 2019 Stackoverflow survey

Community goals

- Passionate about community unification (standardization of method signatures)
- Need to find optimal scalafmt settings
- Strongly dislike UDFs
- Spark tooling?

Spark helper libraries

spark-daria (Scala)

quinn (PySpark)

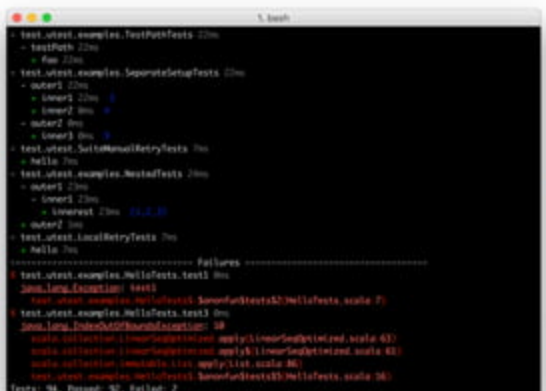
spark-fast-tests / chispa

```
com.github.mrpowers.spark.fast.tests.ColumnMismatch:
+--------------------+-------------+
|                name|expected_name|
+--------------------+-------------+
|               phil|         phil|
|             rashid|       rashid|
|            matthew|        mateo|
|               sami|         sami|
|this is something...|         sami|
|                 li|         feng|
|               null|         null|
+--------------------+-------------+
```

spark-style-guide

Modern Scala libs

uTest



Mill Build Tool

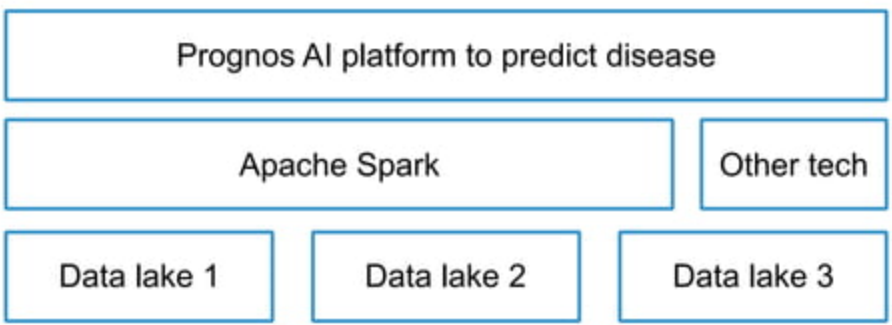
Intro to Mill

Configuring Mill →

Mill is your shiny new Java/Scala build tool! Scared of SBT? Melancholy over Maven? Grumbling about Gradle? Baffled by Bazel? Give Mill a try!

Mill aims for simplicity by re-using concepts you are already familiar with, borrowing ideas from modern tools like Bazel, to let you build your projects in a way that's simple, fast, and predictable.

Prognos data lakes

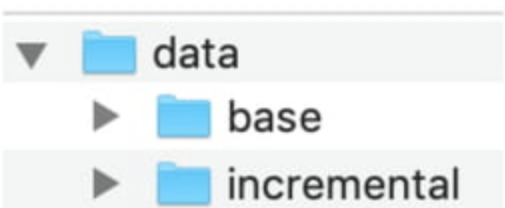


TL;DR

- 1 GB files
- No nested directories

Small file problem

- Incrementally updating a lake will create a lot of small files
- We can store data like this so it's easy to compact



Suppose we have a CSV data lake

- CSV data lake is constantly being updated
- Want to convert it to a Parquet data lake
- Want incremental updates every hour

CSV => Parquet

```
val sDF = spark.readStream
  .schema(schema)
  .csv("/my-cool-bucket/csv-lake/data")

sDF
  .writeStream
  .trigger(Trigger.Once)
  .format("parquet")
  .option("checkpointLocation", "/my-cool-bucket/parquet-lake/checkpoint")
  .start("/my-cool-bucket/parquet-lake/data/incremental")

}
```

Compacting small files

10,000 incremental files and 166GB of data

```
val df = spark
  .read
  .parquet("/my-cool-bucket/parquet-lake/data/incremental")

df
  .coalesce(166)
  .write
  .mode(SaveMode.Append)
  .parquet("/my-cool-bucket/parquet-lake/data/base")
```

Access data lake

```
spark
  .read
  .parquet("/my-cool-bucket/parquet-lake/data/{incremental,base}")
```

Why does the repartition() method increase file size on disk?

11

1

A data lake I am working with (`df`) has 2 TB of data and 20,000 files. I'd like to compact the data set into 2,000 1 GB files.

If you run `df.coalesce(2000)` and write out to disk, the data lake contains 1.9 TB of data.

If you run `df.repartition(2000)` and write out to disk, the data lake contains 2.6 TB of data.

Each file in the `repartition()` data lake is exactly 0.3 GB larger than expected (they're all 1.3 GB files instead of 1 GB files).

Why does the `repartition()` method increase the size of the overall data lake?

Why partition data lakes?

- Data skipping
- Massively improve query performance
- I've seen queries run 50-100 times faster on partitioned lakes

Sample data

first_name	last_name	country
Ernesto	Guevara	Argentina
Vladimir	Putin	Russia
Maria	Sharapova	Russia
Bruce	Lee	China
Jack	Ma	China

Filtering unpartitioned lake

```
df
  .where($"country" === "Russia" && $"first_name".startsWith("M"))
  .explain()
```

```
== Physical Plan ==
Project [first_name#12, last_name#13, country#14]
+- Filter (((isnotnull(country#14) && isnotnull(first_name#12)) && (country#14 = Russia)) &&
StartsWith(first_name#12, M))
   +- FileScan csv [first_name#12,last_name#13,country#14]
        Batched: false,
        Format: CSV,
        Location: InMemoryFileIndex[file:/Users/powers/Documents/tmp/blog_data/people.csv],
        PartitionFilters: [],
        PushedFilters: [IsNotNull(country), IsNotNull(first_name), EqualTo(country,Russia),
StringStartsWith(first_name,M)],
        ReadSchema: struct
```

Partitioning the data lake

```
df
  .repartition($"country")
  .write
  .option("header", "true")
  .partitionBy("country")
  .csv("/Users/powers/Documents/tmp/blog_data/partitioned_lake")
```

Partitioned lake on disk

```
partitioned_lake/
  country=Argentina/
    part-00044-c5d2f540-e89b-40c1-869d-f9871b48c617.c000.csv
  country=China/
    part-00059-c5d2f540-e89b-40c1-869d-f9871b48c617.c000.csv
  country=Russia/
    part-00002-c5d2f540-e89b-40c1-869d-f9871b48c617.c000.csv
```

Filtering Partitioned data lake

```
df
  .where($"country" === "Russia" && $"first_name".startsWith("M"))
  .explain()
```

```
== Physical Plan ==
Project [first_name#74, last_name#75, country#76]
+- Filter (isnotnull(first_name#74) && StartsWith(first_name#74, M))
   +- FileScan csv [first_name#74,last_name#75,country#76]
        Batched: false,
        Format: CSV,
        Location: InMemoryFileIndex[file:/Users/powers/Documents/tmp/blog_data/partitioned_lake],
        PartitionCount: 1,
        PartitionFilters: [isnotnull(country#76), (country#76 = Russia)],
        PushedFilters: [IsNotNull(first_name), StringStartsWith(first_name,M)],
        ReadSchema: struct
```

Comparing physical plans

Unpartitioned

```
Project [first_name#12, last_name#13, country#14]

+- Filter (((isnotnull(country#14) && isnotnull(first_name#12))
&& (country#14 = Russia)) && StartsWith(first_name#12, M))

+- FileScan csv [first_name#12,last_name#13,country#14]
      Batched: false,
      Format: CSV,
      Location: InMemoryFileIndex[….],
      PartitionFilters: [],
      PushedFilters: [IsNotNull(country), IsNotNull(first_name),
EqualTo(country,Russia), StringStartsWith(first_name,M)],
      ReadSchema: struct
```

Partitioned

```
Project [first_name#74, last_name#75, country#76]

+- Filter (isnotnull(first_name#74) && StartsWith(first_name#74, M))


+- FileScan csv [first_name#74, last_name#75, country#76]
      Batched: false,
      Format: CSV,
      Location: InMemoryFileIndex[…],
      PartitionCount: 1,
      PartitionFilters: [isnotnull(country#76), (country#76 = Russia)],
      PushedFilters: [IsNotNull(first_name),
StringStartsWith(first_name,M)],
      ReadSchema: struct
```

Directly grabbing the partitions is faster

```
val russiansDF = spark
  .read
  .csv("/Users/powers/Documents/tmp/blog_data/partitioned_lake/country=Russia")

russiansDF.where($"first_name".startsWith("M"))
```

Real partitioned data lake

- Updates every 3 hours
- Has 5 million files
- 15,000 files are being added every day
- Still great for a lot of queries

Creating partitioned lakes (1/3)

```
// each partition is single file
df
  .repartition($"country")
  .write
  .option("header", "true")
  .partitionBy("country")
  .csv("/Users/powers/Documents/tmp/blog_data/partitioned_lake")
```

Creating partitioned lakes (2/3)

```
// tons of files get written out
df
  .write
  .option("header", "true")
  .partitionBy("country")
  .csv("/Users/powers/Documents/tmp/blog_data/partitioned_lake")
```

Creating partitioned lakes (3/3)

```
import org.apache.spark.sql.functions.rand

// max 100 files per partition
df
  .repartition(100, $"country", rand)
  .write
  .option("header", "true")
  .partitionBy("country")
  .csv("/Users/powers/Documents/tmp/blog_data/partitioned_lake")
```

Compacting Delta Lakes

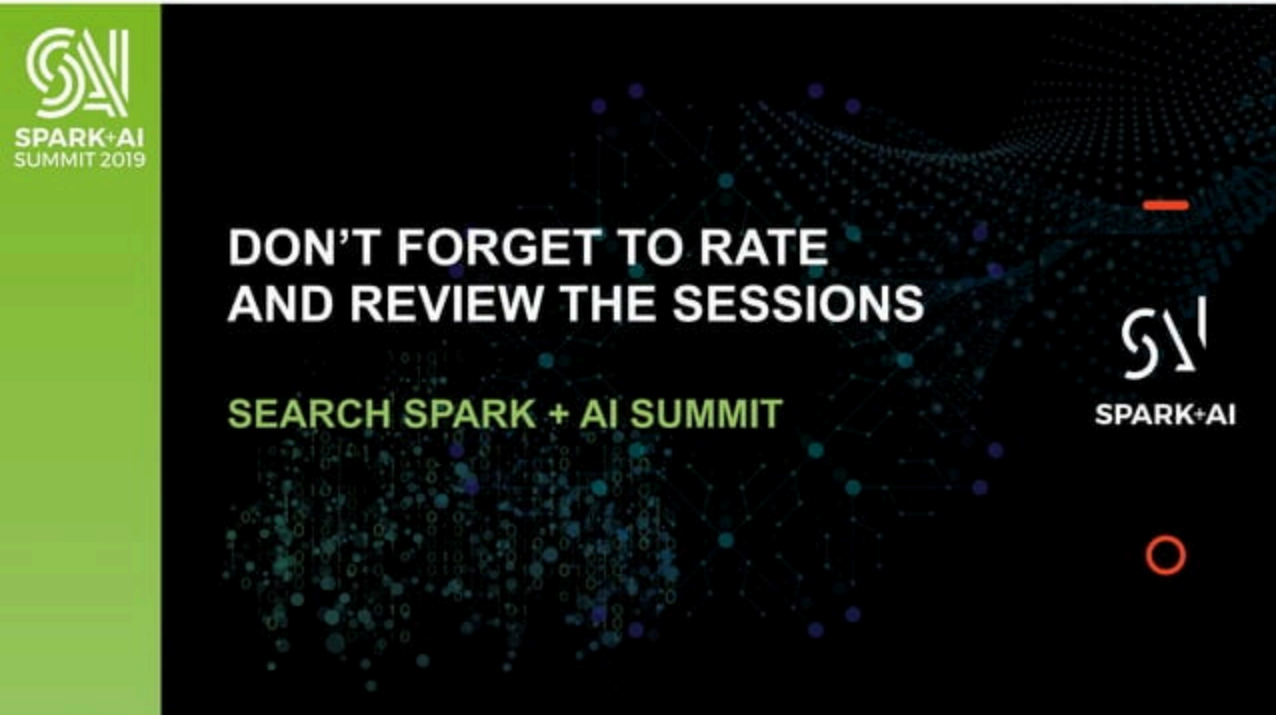
```
// create table
spark.sql("CREATE TABLE delta_pond_for_spike USING DELTA LOCATION '/mnt/some-bucket/delta/pond'")

// compaction
spark.sql("OPTIMIZE delta_pond_for_spike")

// clean up files associated with table
spark.sql("VACUUM delta_pond_for_spike")
```

Incrementally updating partitioned lakes

- Small file problem grows quickly
- Compaction is hard
- Not sure of any automated Parquet compaction algos

What talk should I give next?

- Best practices for the Spark community
- Ditching SBT for the Mill build tool
- Testing Spark code
- Running Spark Scala code in PySpark


SPARK+AI
SUMMIT 2019
DON'T FORGET TO RATE
AND REVIEW THE SESSIONS
SEARCH SPARK + AI SUMMIT
SPARK+AI