Why you should care about data layout in the file system

Cheng Lian, @liancheng
Vida Ha, @femineer
Spark Summit 2017

databricks

About Databricks

TEAM

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

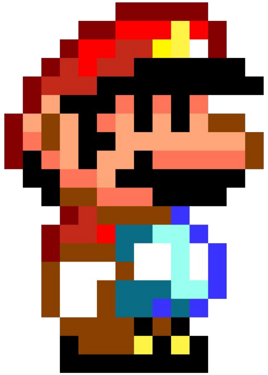
MISSION

Making Big Data Simple

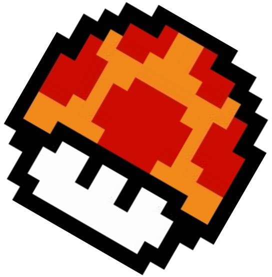
PRODUCT

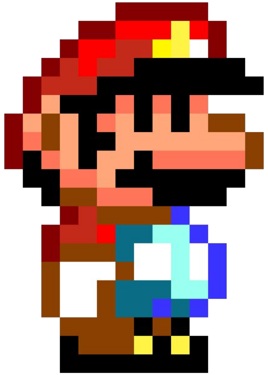
Unified Analytics Platform

Apache Spark is a powerful framework with some temper



Just like
super mario

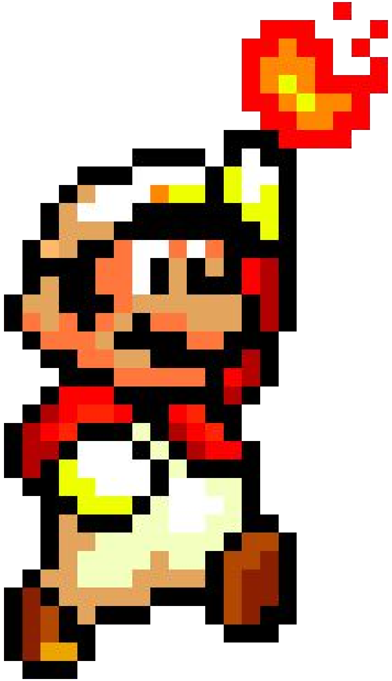




Serve him the
right ingredients

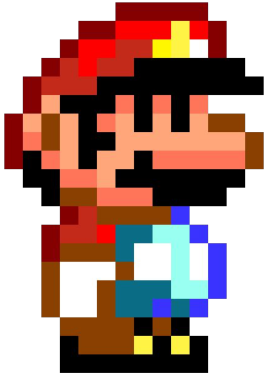
Powers up and
gets more efficient

Keep serving

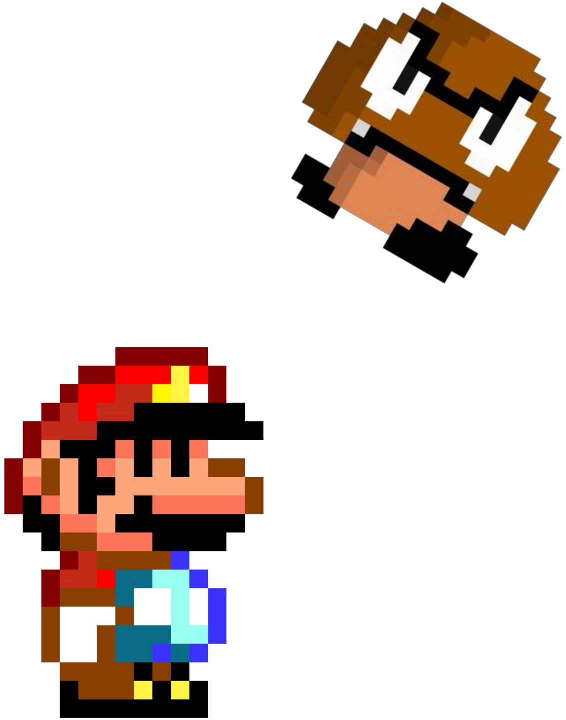


He even knows how to *Spark*!

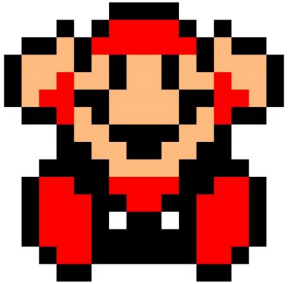
However,
once served
a wrong dish...



Meh...



And sometimes...



It can be messy...

Secret sauces we feed Apache Spark™

File Formats

Choosing a compression scheme

The obvious

- Compression ratio: the higher the better
- De/compression speed: the faster the better

Choosing a compression scheme

Splittable v.s. non-splittable

- Affects parallelism, crucial for big data
- Common splittable compression schemes
  - LZ4, Snappy, BZip2, LZO, and etc.
- *GZip is non-splittable*
  - Still common if file sizes are << 1GB
  - Still applicable for Parquet

Columnar formats

Smart, analytics friendly, optimized for big data

- Support for nested data types
- Efficient data skipping
  - Column pruning
  - Min/max statistics based predicate push-down
- Nice interoperability
- Examples:
  - Spark SQL built-in support: Apache Parquet and Apache ORC
  - Newly emerging: Apache CarbonData and Spinach

Columnar formats

Parquet

- Apache Spark default output format
- Usually the best practice for Spark SQL
- Relatively heavy write path
  - Worth the time to encode for repeated analytics scenario
- Does not support fine grained appending
  - Not ideal for, e.g., collecting logs
- Check out Parquet presentations for more details

Semi-structured text formats

Sort of structured but not self-describing

- Excellent write path performance but slow on the read path
  - Good candidates for collecting raw data (e.g., logs)
- Subject to inconsistent and/or malformed records
- Schema inference provided by Spark (for JSON and CSV)
  - Sampling-based
  - Handy for exploratory scenario but can be inaccurate
  - Always specify an accurate schema in production

Semi-structured text formats

JSON

- Supported by Apache Spark out of the box
- One JSON object per line for fast file splitting
- JSON object: map or struct?
  - Spark schema inference always treats JSON objects as structs
  - Watch out for arbitrary number of keys (may OOM executors)
  - Specify an accurate schema if you decide to stick with maps

Semi-structured text formats

JSON

- Malformed records
  - Bad records are collected into column `_corrupted_record`
  - All other columns are set to null

Semi-structured text formats

CSV

- Supported by Spark 2.x out of the box
  - Check out the spark-csv package for Spark 1.x
- Often used for handling legacy data providers & consumers
  - Lacks of a standard file specification
    - Separator, escaping, quoting, and etc.
  - Lacks of support for nested data types

Raw text files

Arbitrary line-based text files

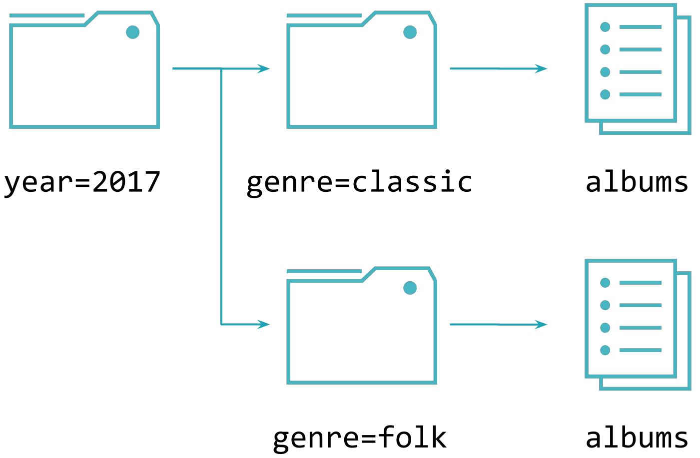
- Splitting files into lines using `spark.read.text()`
  - Keep your lines a reasonable size
- Keep file size < 1GB if compressed with a non-splittable compression scheme (e.g., GZip)
- Handing inevitable malformed data
  - Use a `filter()` transformation to drop bad lines, or
  - Use a `map()` transformation to fix bad line

Directory layout

Partitioning

Overview

- Coarse-grained data skipping
- Available for both persisted tables and raw directories
- Automatically discovers Hive style partitioned directories



databricks

Partitioning

SQL

```
CREATE TABLE ratings
USING PARQUET
PARTITIONED BY (year, genre)
AS SELECT artist, rating, year, genre
FROM music
```

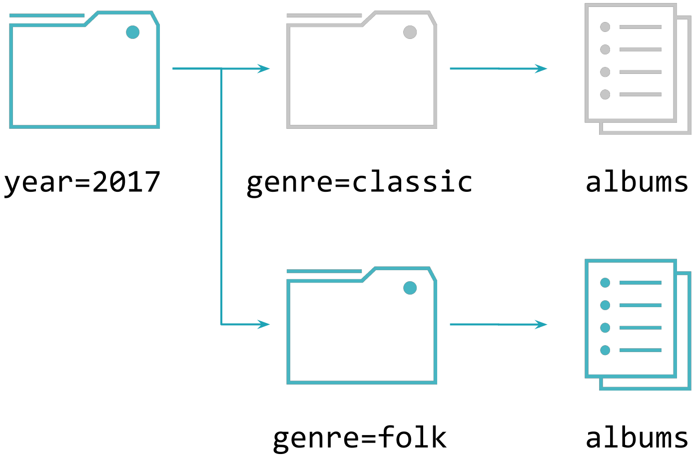
DataFrame API

```
spark
  .table(“music”)
  .select(’artist, ’rating, ’year, ’genre)
  .write
  .format(“parquet”)
  .partitionBy(’year, ’genre)
  .saveAsTable(“ratings”)
```

Partitioning

Filter predicates

Use simple filter predicates containing partition columns to leverage partition pruning



Partitioning

Filter predicates

- year = 2000 AND genre = ‘folk’
- year > 2000 AND rating > 3
- year > 2000 OR genre <> ‘rock’

Partitioning

Filter predicates

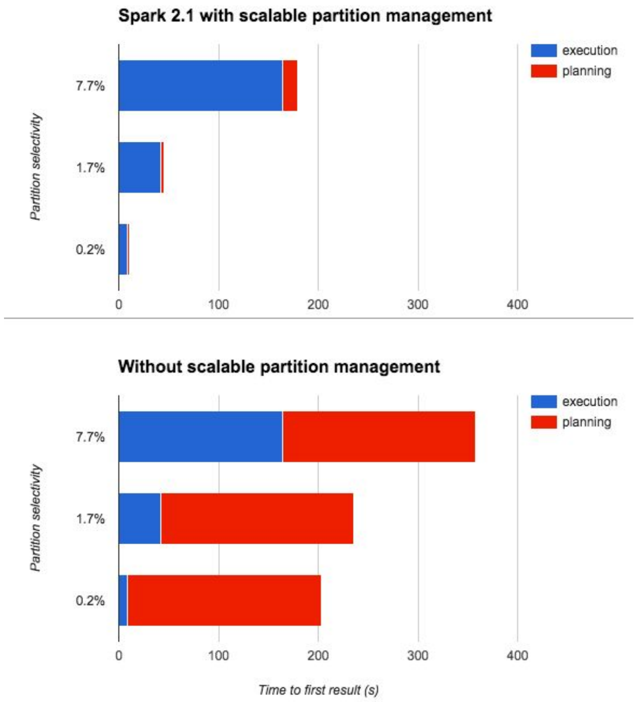
- `year > 2000 OR rating = 5`
- `year > rating`

Partitioning

Avoid excessive partitions

- Stress metastore for persisted tables
- Stress file system when reading directly from the file system
- Suggestions
  - Avoid using too many partition columns
  - Avoid using partition columns with too many distinct values
    - Try hashing the values
    - E.g., partition by first letter of first name rather than first name

Partitioning



Scalable partition handling

Using persisted partitioned tables with Spark 2.1+

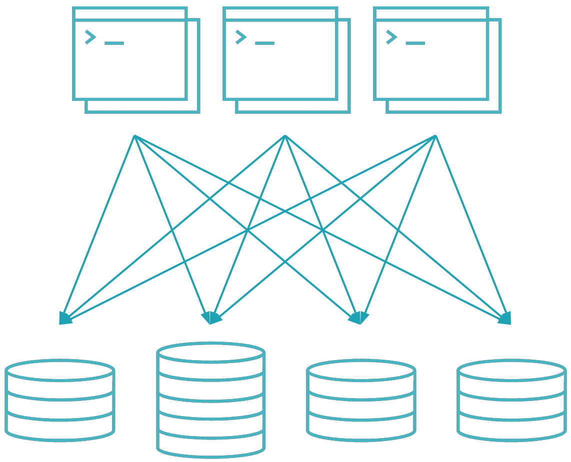
- Per-partition metadata gets persisted into the metastore
- Avoids unnecessary partition discovery (esp. valuable for S3)

Check our blog post for more details

Bucketing

Overview

- *Pre-shuffles* and optionally *pre-sorts* the data while writing
- Layout information gets persisted in the metastore
- Avoids shuffling and sorting when joining large datasets
- Only available for persisted tables



Bucketing

SQL

```
CREATE TABLE ratings
USING PARQUET
PARTITIONED BY (year, genre)
CLUSTERED BY (rating) INTO 5 BUCKETS
SORTED BY (rating)
AS SELECT artist, rating, year, genre
FROM music
```

DataFrame

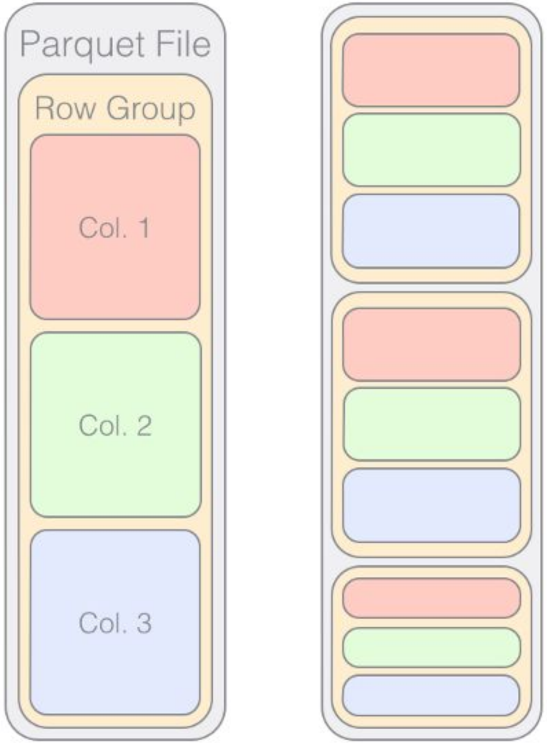
```
ratings
  .select(’artist, ’rating, ’year, ’genre)
  .write
  .format(“parquet”)
  .partitionBy(“year”, “genre”)
  .bucketBy(5, “rating”)
  .sortBy(“rating”)
  .saveAsTable(“ratings”)
```

Bucketing

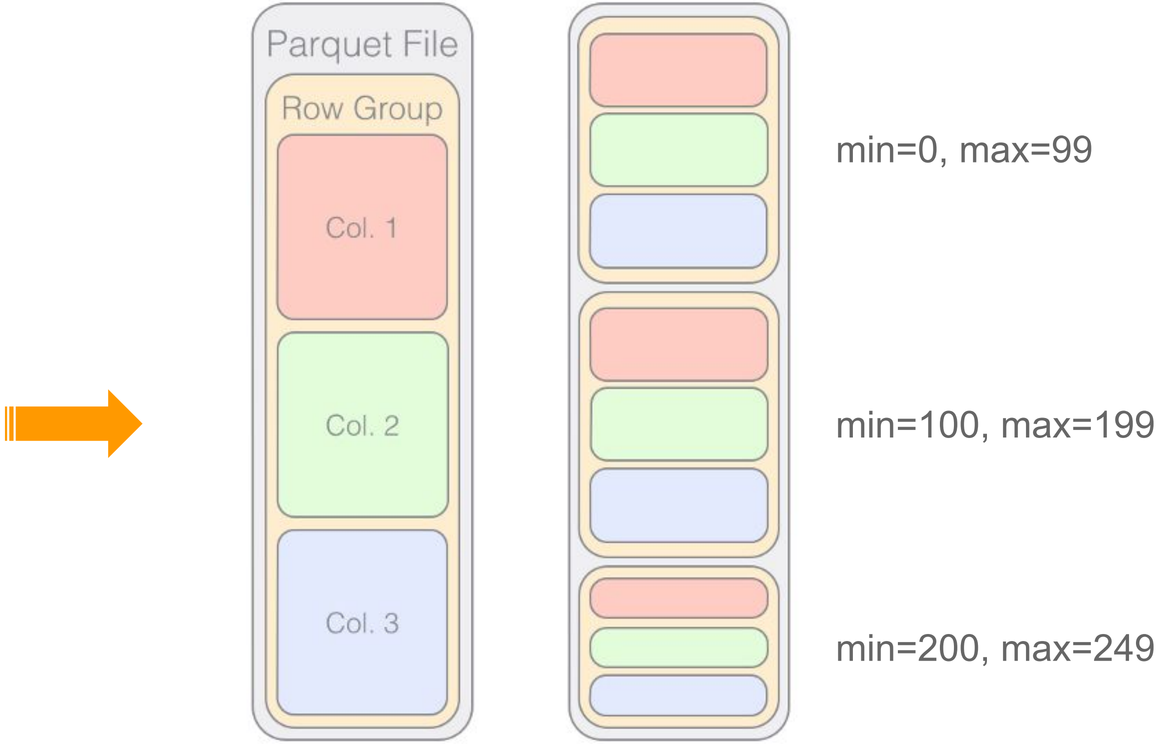
In combo with columnar formats

- Bucketing
  - Per-bucket sorting
- Columnar formats
  - Efficient *data skipping* based on min/max statistics
  - Works best when the searched columns are *sorted*

Bucketing



Bucketing


Parquet File
Row Group
Col. 1
Col. 2
Col. 3
min=0, max=99
min=100, max=199
min=200, max=249

Bucketing

In combo with columnar formats

Perfect combination, makes your Spark jobs FLY!

More tips

File size and compaction

Avoid small files

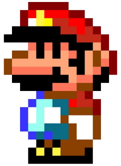
- Cause excessive parallelism
  - Spark 2.x improves this by packing small files
- Cause extra file metadata operations
  - Particularly bad when hosted on S3

File size and compaction

How to control output file sizes

- In general, one task in the output stage writes one file
  - Tune parallelism of the output stage
- `coalesce(N)`, for
  - Reduces parallelism for small jobs
- `repartition(N)`, for
  - Increasing parallelism for all jobs, or
  - Reducing parallelism of final output stage for large jobs
  - Still preserves high parallelism for previous stages

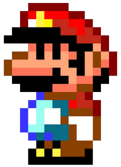
True story



Customer

- Spark ORC Read Performance is much slower than Parquet
- The same query took
  - 3 seconds on a Parquet dataset
  - 4 minutes on an equivalent ORC dataset

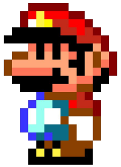
True story

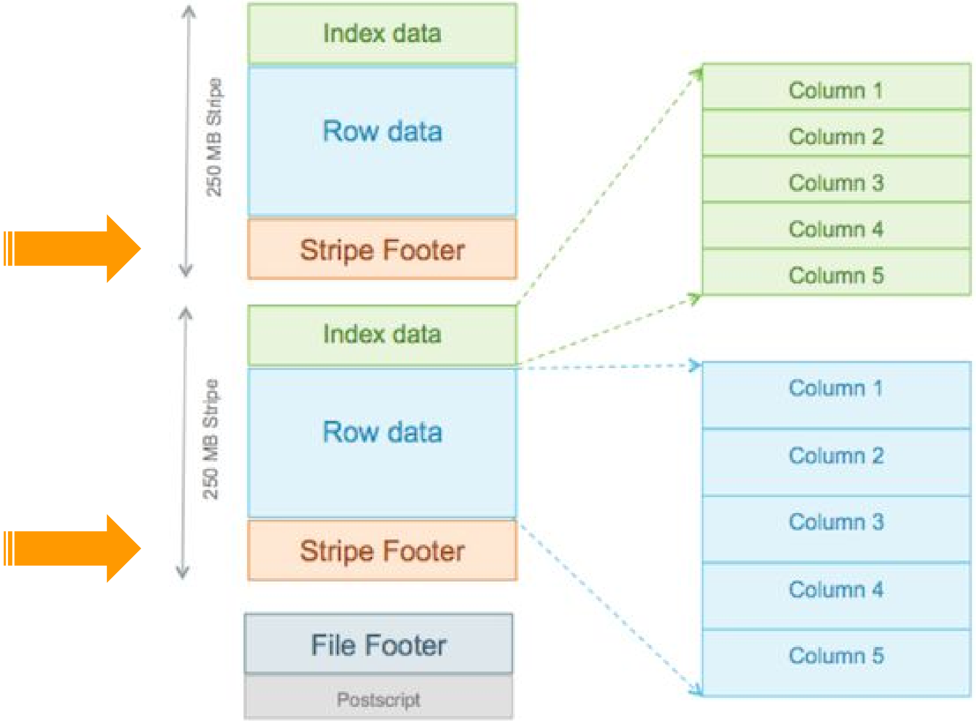


Me

- Ran a simple `count(*)`, which took
  - Seconds on the Parquet dataset with a handful IO requests
  - 35 minutes on the ORC dataset with 10,000s of IO requests
- Most task execution threads are *reading ORC stripe footers*

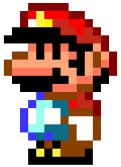
True story





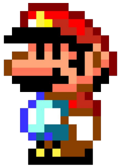
databricks

True story



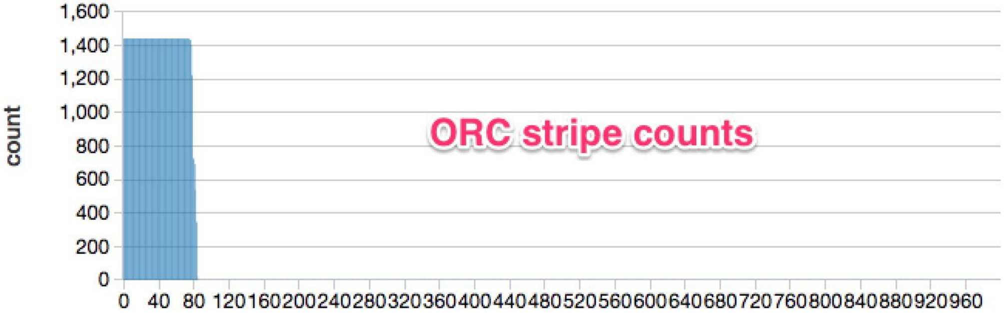
```
import org.apache.hadoop.hive.ql.io.orc._
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.fs.Path

val conf = new Configuration

def countStripes(file: String): Int = {
  val path = new Path(file)
  val reader = OrcFile.createReader(path, OrcFile.readerOptions(conf))
  val metadata = reader.getMetadata
  metadata.getStripeStatistics.size
}
```

True story

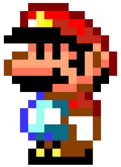


Maximum file size: ~15 MB

Maximum ORC stripe counts: ~1,400



True story

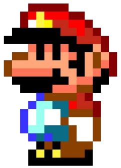


Root cause

Malformed (but not corrupted) ORC dataset

- ORC readers read the footer first before consuming a strip
- ~1,400 stripes within a single file as small as 15 MB
- ~1,400 x 2 read requests issued to S3 for merely 15 MB of data

True story



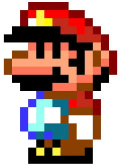
Root cause

Malformed (but not corrupted) ORC dataset

- ORC readers read the footer first before consuming a strip
- ~1,400 stripes within a single file as small as 15 MB
- ~1,400 x 2 read requests issued to S3 for merely 15 MB of data

Much worse than even CSV, not mention Parquet

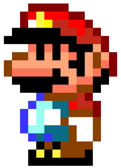
True story



Why?

- Tiny ORC files (~10 KB) generated by Streaming jobs
  - Resulting one tiny ORC stripe inside each ORC file
  - The footers might take even more space than the actual data!

True story



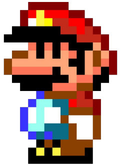
Why?

Tiny files got compacted into larger ones using

```
ALTER TABLE ... PARTITION (...) CONCATENATE;
```

The CONCATENATE command just, well, *concatenated* those tiny stripes and produced larger (~15 MB) files with a huge number of tiny stripes.

True story

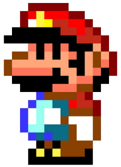


Lessons learned

Again, avoid writing small files in *columnar formats*

- Output files using CSV or JSON for Streaming jobs
    - For better write path performance
- Compact small files into large chunks of columnar files later
    - For better read path performance

True story



The cure

Simply read the ORC dataset and write it back using

```
spark.read.orc(input).write.orc(output)
```

So that stripes are adjusted into more reasonable sizes.

Schema evolution

Columns come and go

- Never ever change the data type of a published column
- Columns with the same name should have the same data type
- If you really dislike the data type of some column
    - Add a new column with a new name and the right data type
    - Deprecate the old one
    - Optionally, drop it after updating all downstream consumers

Schema evolution

Columns come and go

Spark built-in data sources that support schema evolution

- JSON
- Parquet
- ORC

Schema evolution

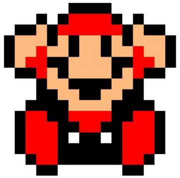
Common columnar formats are less tolerant of data type mismatch. E.g.:

- `INT` cannot be promoted to `LONG`
- `FLOAT` cannot be promoted to `DOUBLE`

JSON is more tolerating, though

- `LONG` → `DOUBLE` → `STRING`

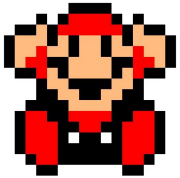
True story



Customer

Parquet dataset corrupted!!! HALP!!!

True story



What happened?

Original schema

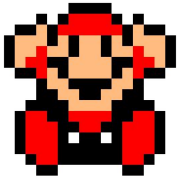
- `{col1: DECIMAL(19, 4), col2: INT}`

Accidentally appended data with schema

- `{col1: DOUBLE, col2: DOUBLE}`

All files written into the same directory

True story



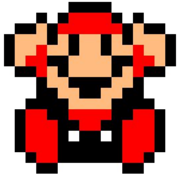
What happened?

Common columnar formats are less tolerant of data type mismatch. E.g.:

- `INT` cannot be promoted to `LONG`
- `FLOAT` cannot be promoted to `DOUBLE`

Parquet considered these schemas as incompatible ones and refused to merge them.

True story



BTW

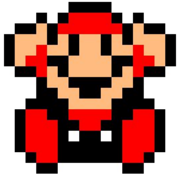
JSON schema inference is more tolerating

- `LONG` → `DOUBLE` → `STRING`

However

- JSON is NOT suitable for analytics scenario
- Schema inference is unreliable, not suitable for production

True story



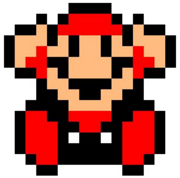
The cure

Correct the schema

- Filter out all the files with the wrong schema
- Rewrite those files using the correct schema

Exhausting because all files are appended into a single directory

True story



Lessons learned

- Be very careful on the write path
- Consider partitioning when possible
    - Better read path performance
    - Easier to fix the data when something went wrong

Recap

File formats

- Compression schemes
- Columnar (Parquet, ORC)
- Semi-structured (JSON, CSV)
- Raw text format

Directory layout

- Partitioning
- Bucketing

Other tips

- File sizes and compaction
- Schema evolution

Try Apache Spark in Databricks!

UNIFIED ANALYTICS PLATFORM

- Collaborative cloud environment
- Free version (community edition)

DATABRICKS RUNTIME 3.0

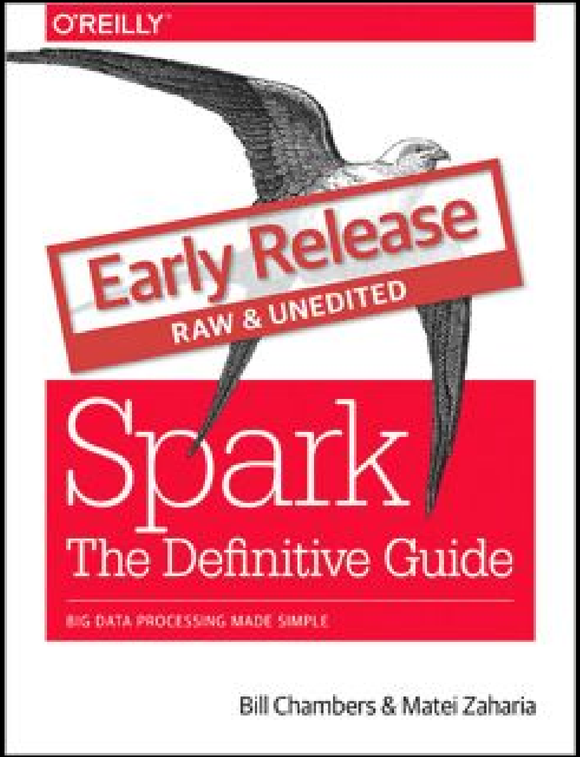
- Apache Spark - optimized for the cloud
- Caching and optimization layer - DBIO
- Enterprise security - DBES

Try for free today.

databricks.com

Early draft available for free today!

go.databricks.com/book



Thank you

Q & A

databricks