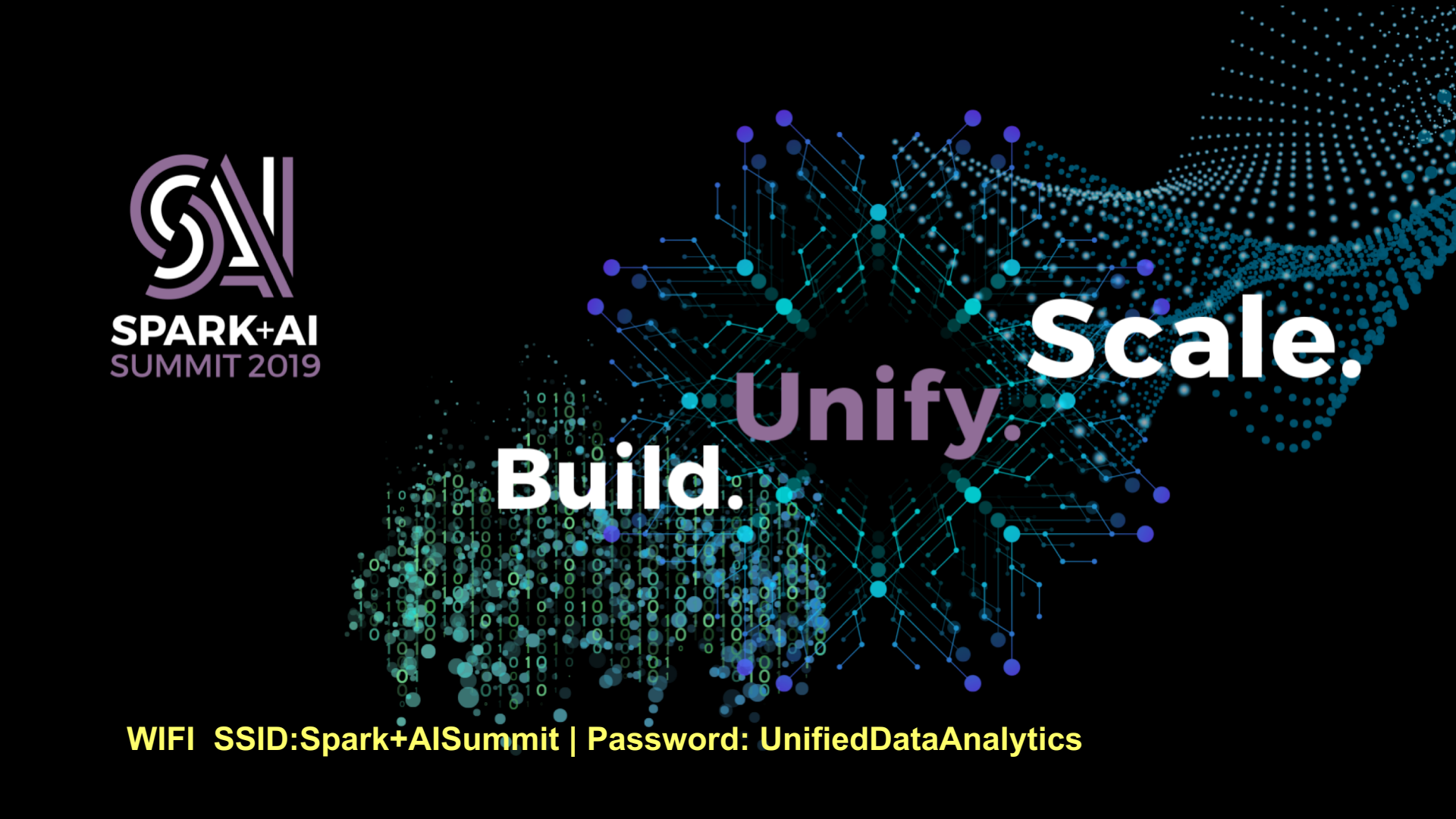


WIFI SSID:Spark+AISummit | Password: UnifiedDataAnalytics

Apache Spark's Built-in File Sources in Depth

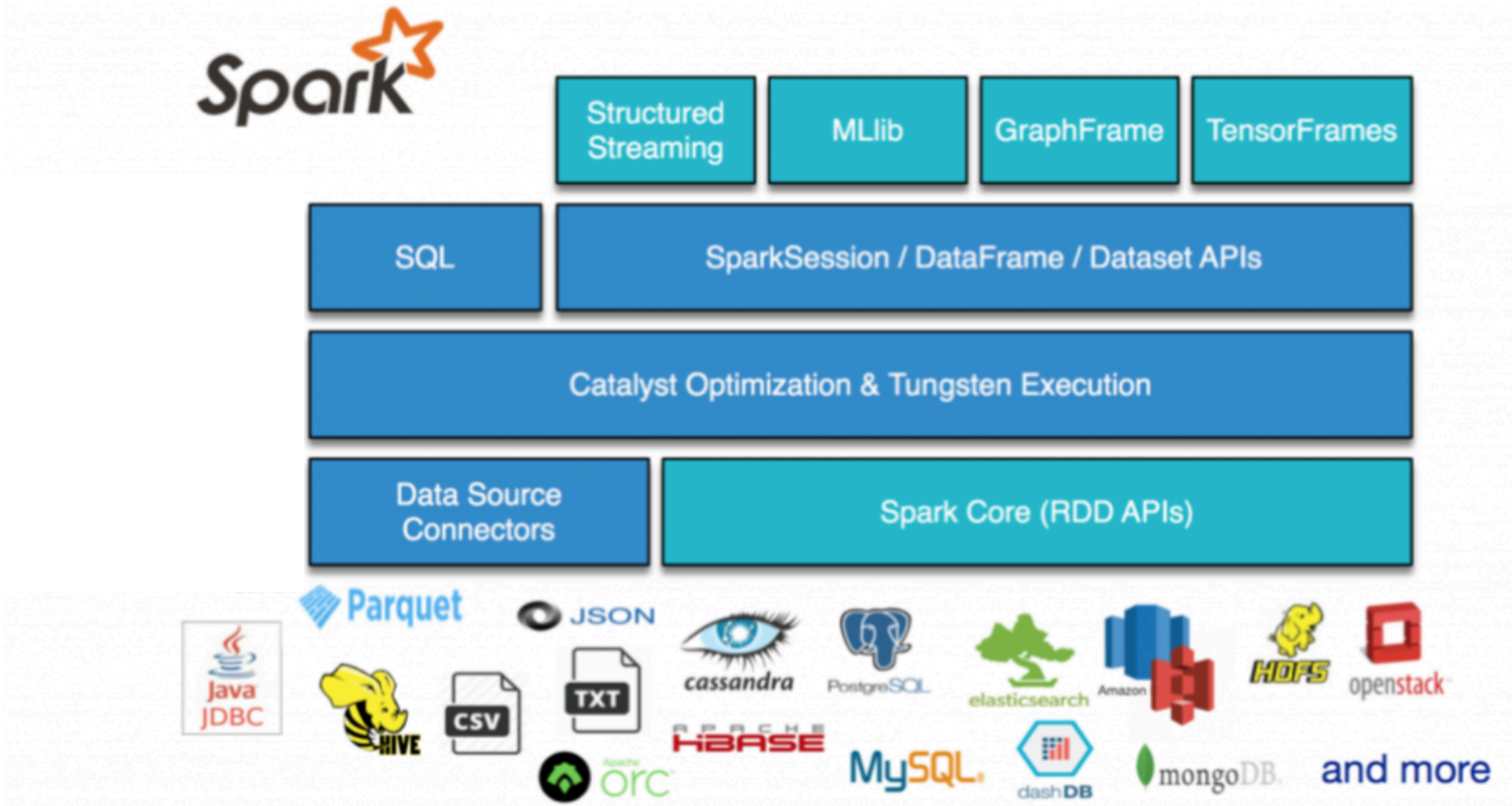
Gengliang Wang, Databricks

About me

- Gengliang Wang(Github:gengliangwang)
- Software Engineer at databricks
- Apache Spark committer

Agenda

- **File formats**
- Data layout
- File reader internals
- File writer internals


Spark
Structured Streaming
MLlib
GraphFrame
TensorFrames
SQL
SparkSession / DataFrame / Dataset APIs
Catalyst Optimization & Tungsten Execution
Data Source Connectors
Spark Core (RDD APIs)
Parquet
JSON
Java JDBC
HIVE
CSV
TXT
cassandra
PostgreSQL
elasticsearch
Amazon
HDFS
openstack
HBASE
orc
MySQL
dashDB
mongoDB.
and more

Built-in File formats

- Column-oriented
  - Parquet
  - ORC
- Row-oriented
  - Avro
  - JSON
  - CSV
  - Text
  - Binary

Column vs. Row

Example table “music”

artist	genre	year	album
Elvis Presley	Country	1965	Love Me Tender
The Beatles	Rock	1966	Revolver
Nirvana	Rock	1991	Nevermind

Column vs. Row

Column

Elvis Presley, The Beatles, Nirvana

Country, Rock, Rock

1965, 1966, 1991

Love Me Tender, Revolver, Nevermind

Row

Elvis Presley, Country, 1965, Love Me Tender

The Beatles, Rock, 1966, Revolver

Nirvana, Rock, 1991, Nevermind

artist genre

year album

Column-oriented formats

Column

Elvis Presley, The Beatles, Nirvana

Country, Rock, Rock

1965, 1966, 1991

Love Me Tender, Revolver, Nevermind

```
SELECT album
FROM music
WHERE artist = 'The Beatles';
```

Artist Genre
Year Album

Row-oriented formats

```
SELECT * FROM music;

INSERT INTO music ...
```

Row

Elvis Presley, Country, 1965, Love Me Tender

The Beatles, Rock, 1966, Revolver

Nirvana, Rock, 1991, Nevermind

artist genre

year album

Column vs. Row

Column

- Read heavy workloads
- Queries that require only a subset of the table columns

Row

- Write heavy workloads
  - event level data
- Queries that require most or all of the table columns

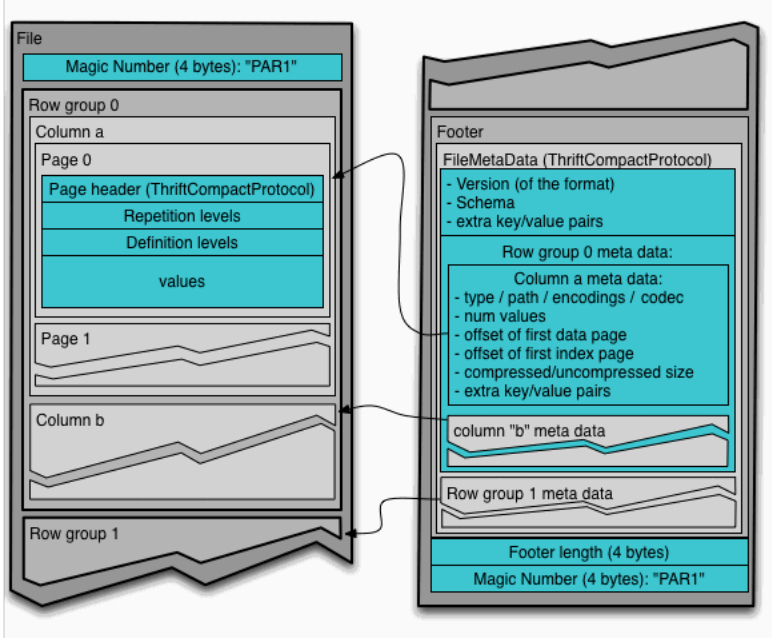
Built-in File formats

- Column-oriented
  - Parquet
  - ORC
- Row-oriented
  - Avro
  - JSON
  - CSV
  - Text
  - Binary

Parquet

File Structure

- A list of row groups
- Footer
  - Metadata of each row group
    - Min/max of each column
  - Schema



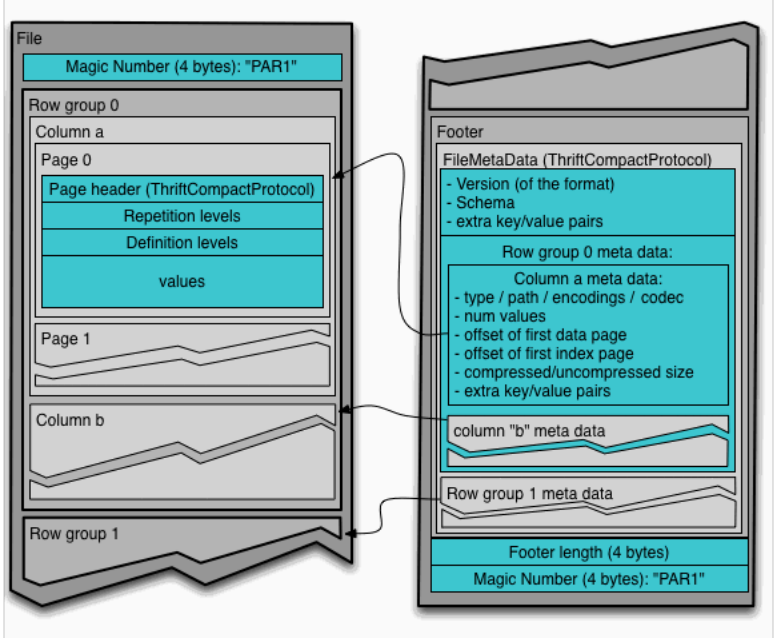
Parquet

Row group

- A list of column trunks
  - One column chunk per column

Column chunk

- A list of data pages
- An optional dictionary page



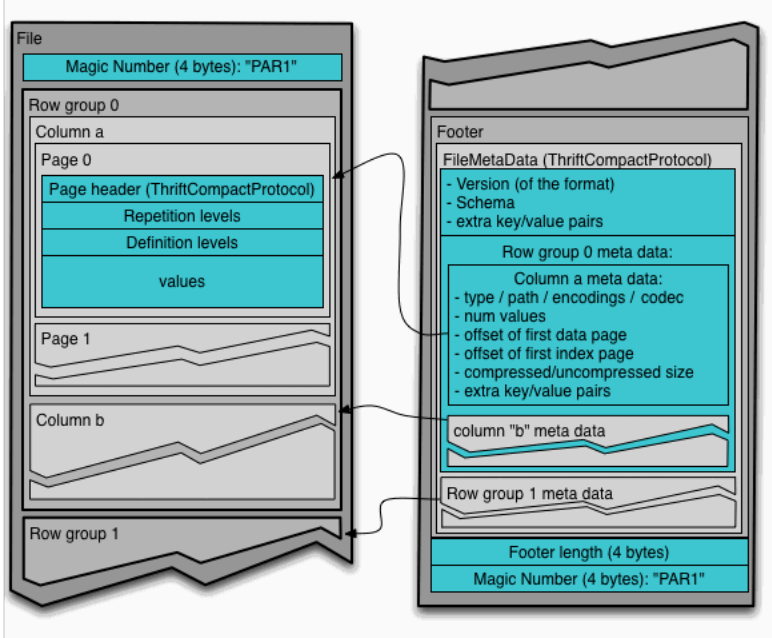
Parquet

Predicates pushed down

- a = 1
- a < 1
- a > 1
- …

Row group skipping

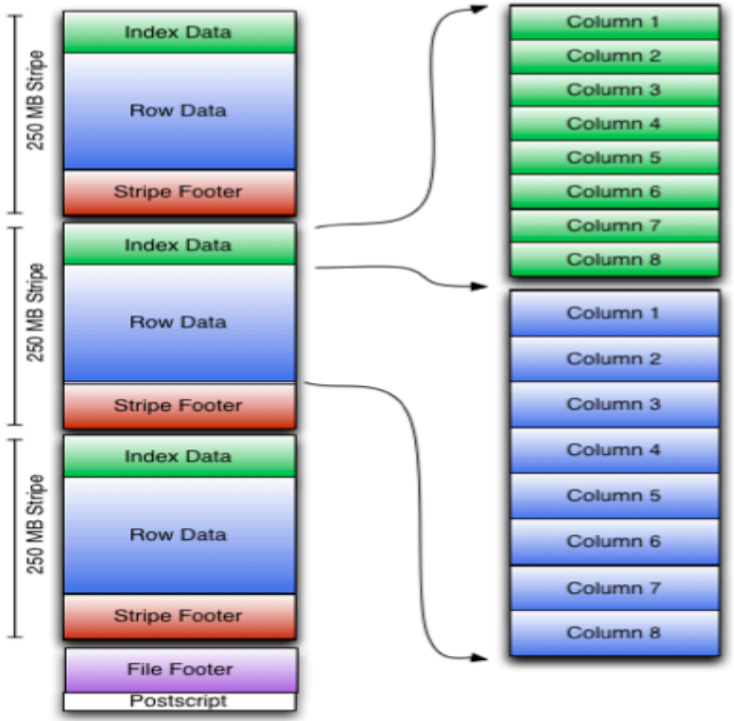
- Footer stats (min/max)
- Dictionary pages of column chunks



ORC

File Structure

- A list of stripes
- Footer
  - column-level aggregates count, min, max, and sum
  - The number of rows per stripe

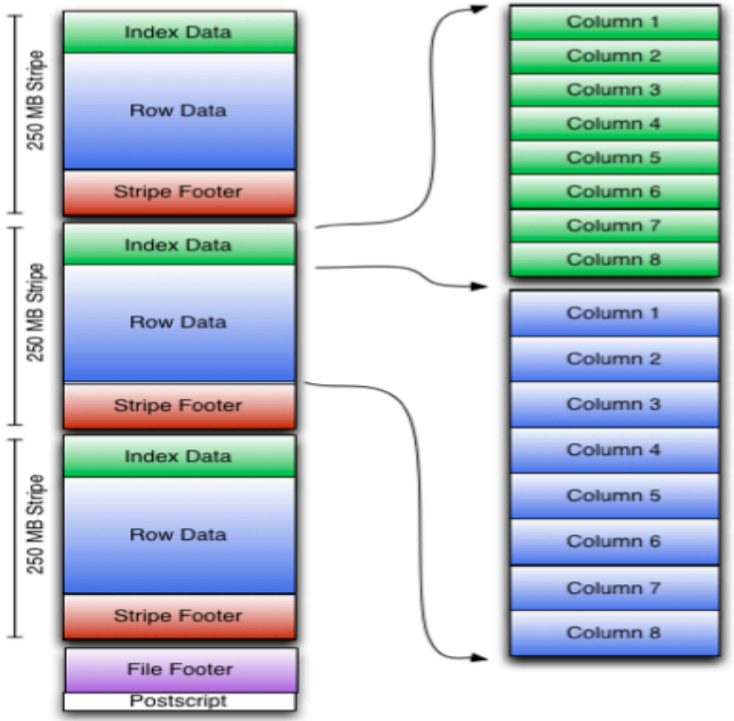


ORC

Stripe Structure

- Indexes
  - Start positions of each column
  - Min/max of each column
- Data

Supports Stripe skipping



Built-in File formats

- Column-oriented
  - Parquet
  - ORC
- Row-oriented
  - Avro
  - JSON
  - CSV
  - Text
  - Binary

Avro

- Compact
- Fast
- Robust support for schema evolution
  - Adding fields
  - Removing fields
  - Renaming fields(with aliases)
  - Change default values
  - Change docs of fields

Semi-structured text formats: JSON & CSV

- 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
  - Sampling-based
  - Handy for exploratory scenario but can be inaccurate

JSON

- 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

Avro vs. JSON

Avro

- Compact and Fast
- Accurate schema inference
- High data quality and robust schema revolution support, good choice for organizational scale development
  - Event data from Web/iOS/Android clients

JSON

- Repeating every field name with every single record
- Schema inference can inaccurate
- Light weight, easy development and debug

CSV

- 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
- Limited schema with only one field
  - value: (StringType)

Binary

- New file data source in Spark 3.0
- Reads binary files and converts each file into a single record that contains the raw content and metadata of the file

Binary

Schema

- path (StringType)
- modificationTime (TimestampType)
- length (LongType)
- content (BinaryType)

Binary

To reads all JPG files recursively from the input directory and ignores partition discovery

```
spark.read.format("binaryFile")
  .option("pathGlobFilter", "*.jpg")
  .option("recursiveFileLookup", "true")
  .load("/path/to/dir")
```

Agenda

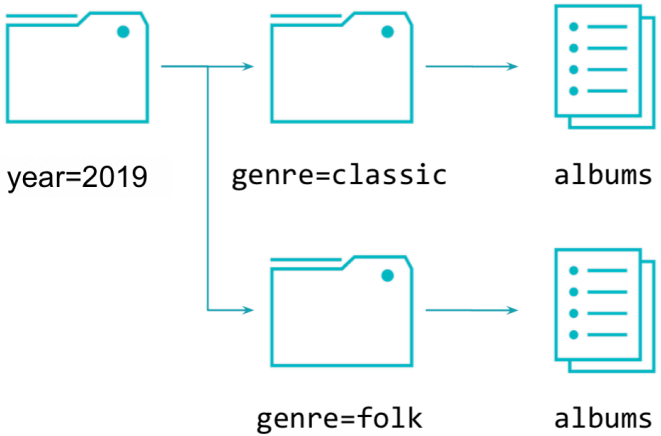
- File formats
- **Data layout**
- File reader internals
- File writer internals

Data layout

- Partitioning
- Bucketing

Partitioning

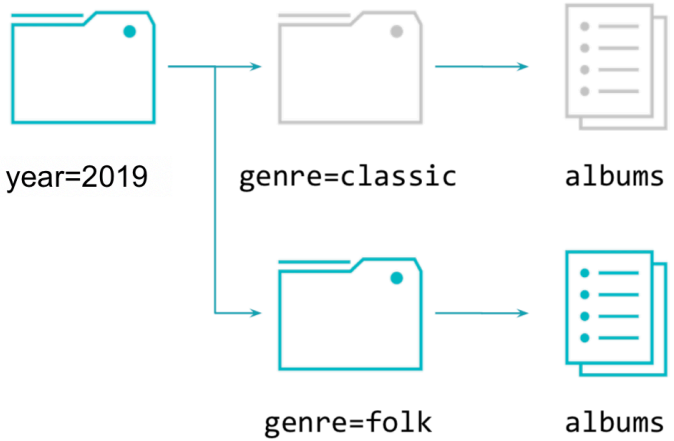
- Coarse-grained data skipping
- Available for both persisted tables and raw directories
- Automatically discovers Hive style partitioned directories



Partitioning

Filter predicates

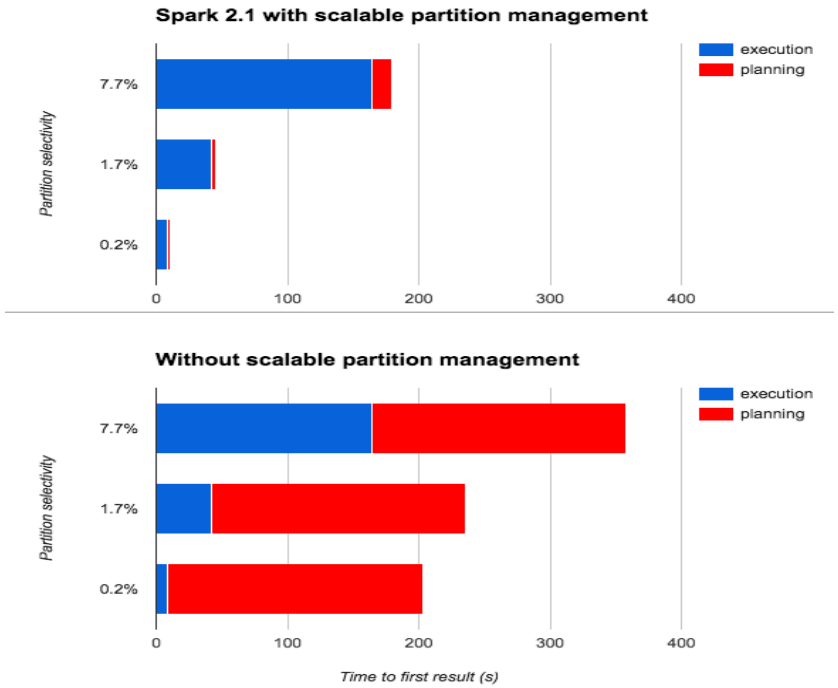
- Use simple filter predicates containing partition columns to leverage partition pruning
- Avoids unnecessary partition discovery (esp. valuable for S3)



SELECT * FROM music WHERE
year = 2019 and genre='folk'

Partitioning

Check our blog post https://tinyurl.com/y2eow2rx for more details



Partitioning

SQL

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

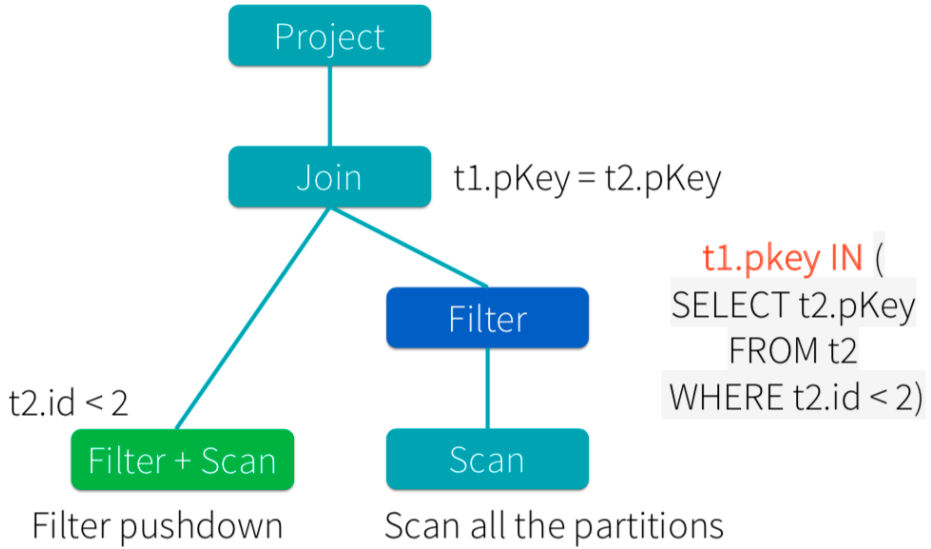
DataFrame API

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

Partitioning: new feature in Spark 3.0

Dynamic Partition Pruning

```
SELECT t1.id, t2.pKey
FROM   t1
  JOIN t2
    ON t1.pKey = t2.pKey
    AND t2.id < 2
```



Partitioning: new feature in Spark 3.0

More details in the talk

Dynamic Partition Pruning in Apache Spark

Bogdan Ghit & Juliusz Sompolski

Partitioning

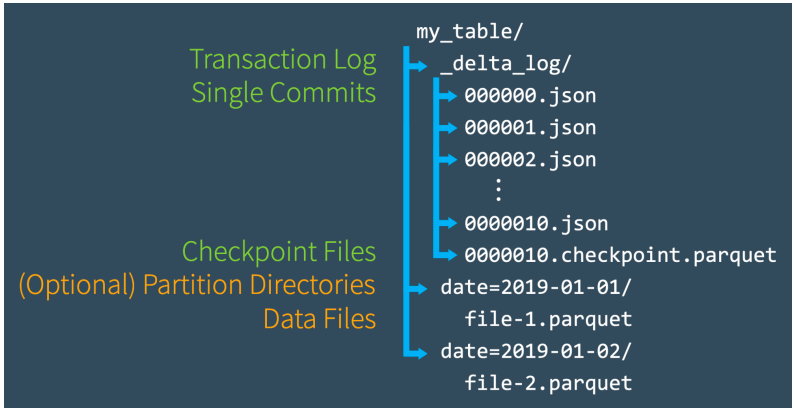
Avoid excessive partitions

- Stress metastore for getting partition columns and values
- Stress file system for listing files under partition directories
  - Esp. slow on S3
- 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
  - Try Delta Lake

Partitioning

Delta Lake's transaction logs are analyzed in parallel by Spark

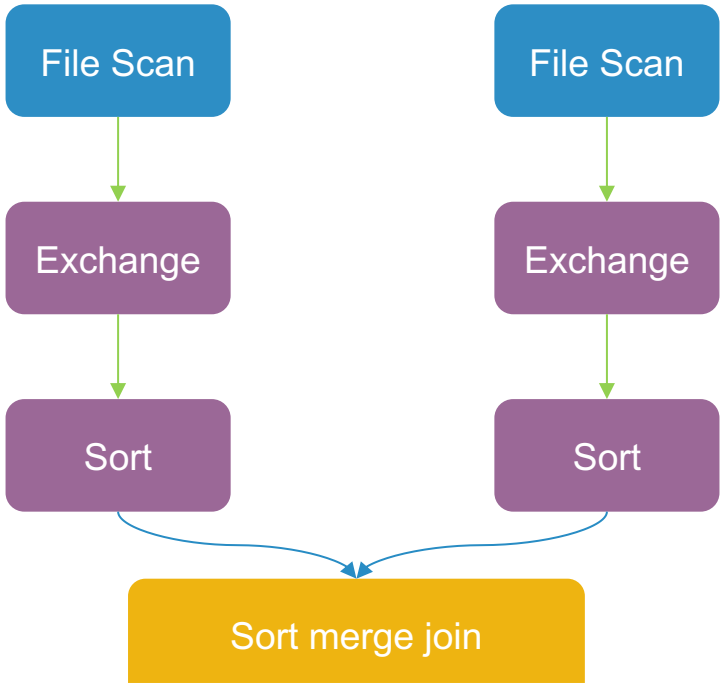
- No need to stress Hive Metastore for partition values and file system for listing files
- Able to handle billions of partitions and files



Data layout

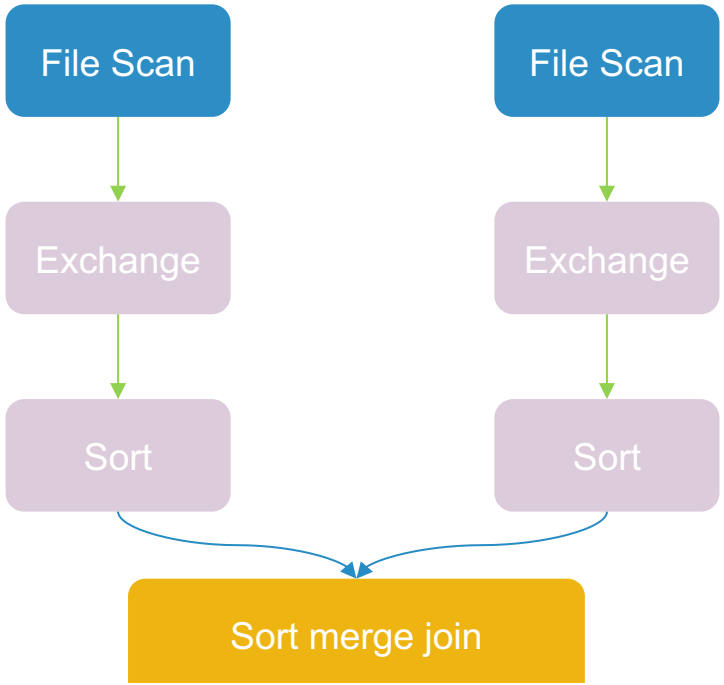
- Partitioning
- **Bucketing**

Bucketing



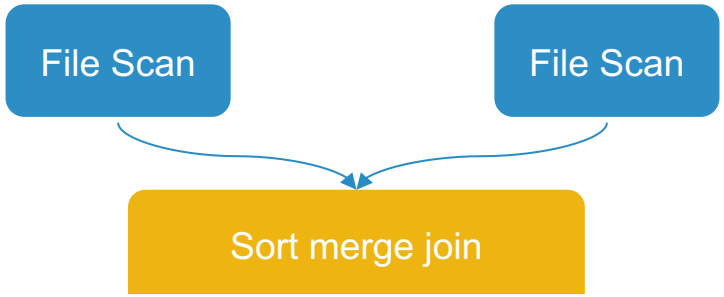
Shuffle is slow.

Bucketing



Pre-shuffle
+
Pre-sort

Bucketing



Use bucketing on tables which are frequently JOINed with same keys

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 API

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

Bucketing

In combo with columnar formats: works best when the searched columns are *sorted*

- Bucketing
  - Per-bucket sorting
- Columnar formats
  - Efficient *data skipping* based on min/max statistics

Agenda

- File formats
- Data layout
- **File reader internals**
- File writer internals

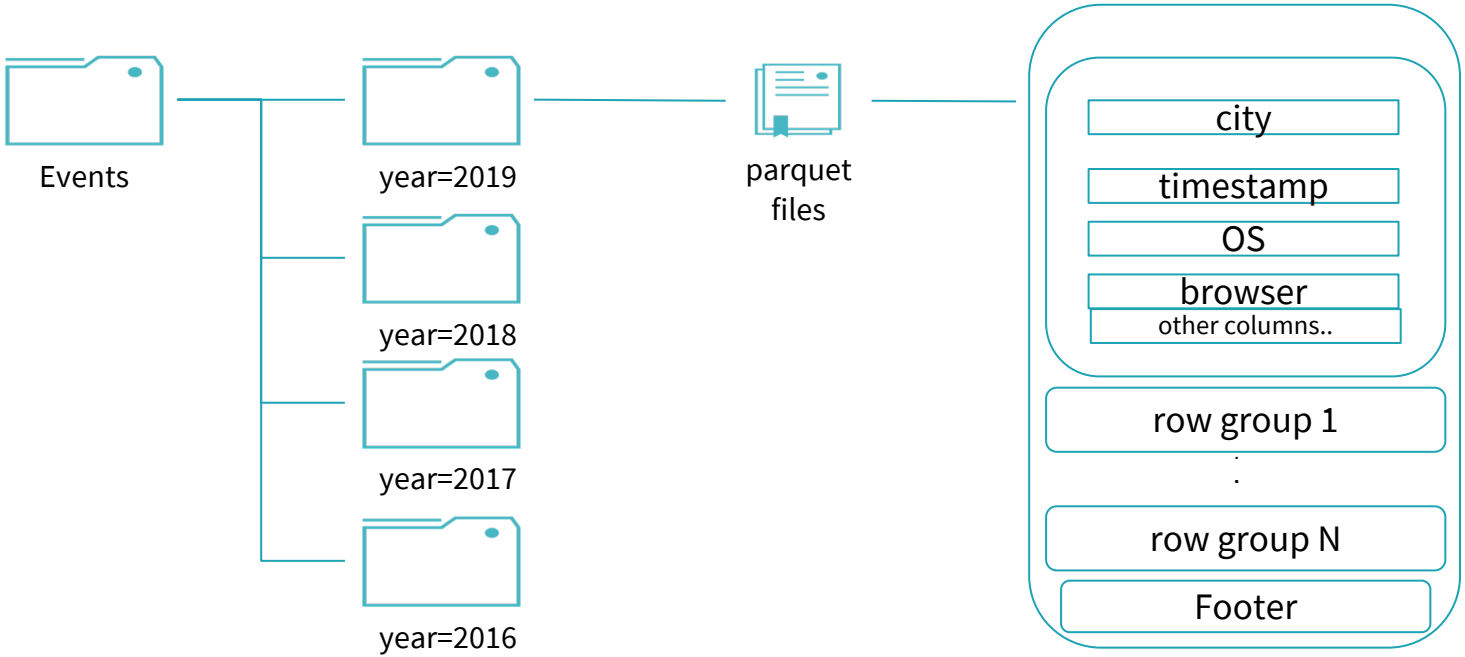
Query example

```
spark.read.parquet("/data/events")
  .where("year = 2019")
  .where("city = 'Amsterdam'")
  .select("timestamp")
```

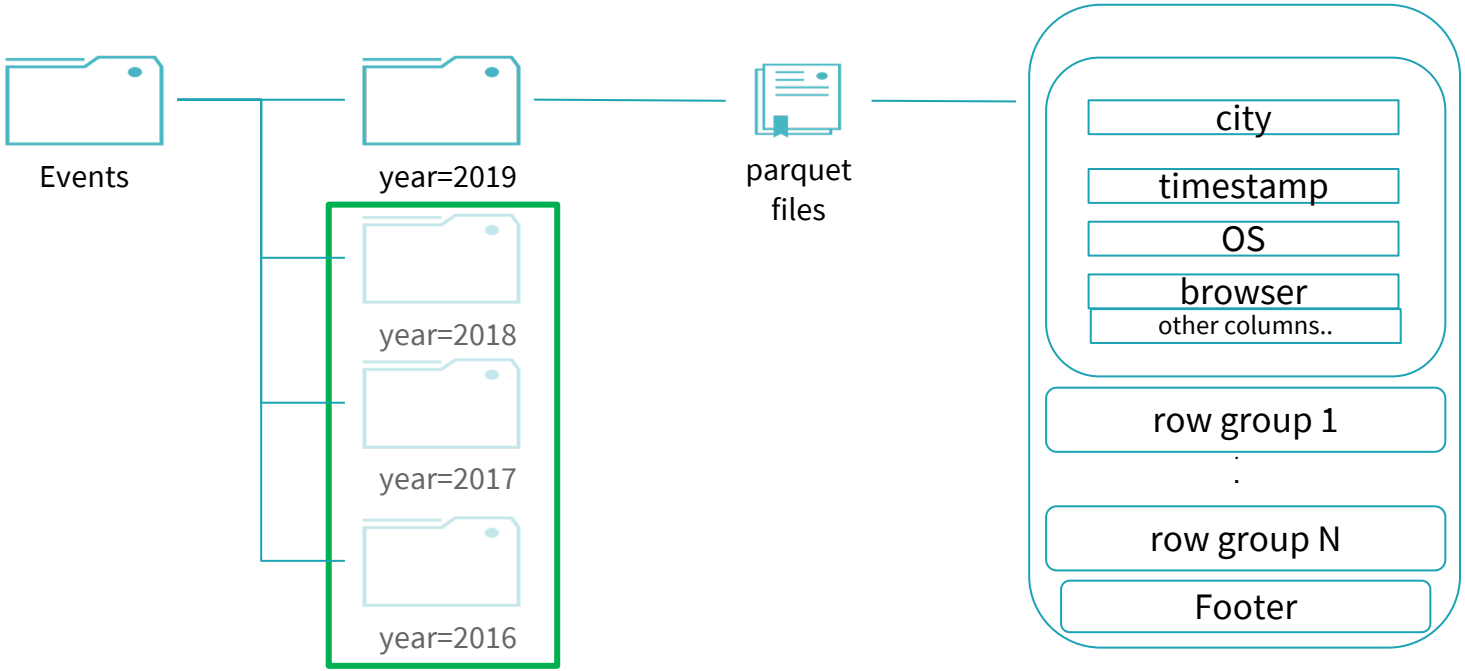
Goal

- Understand data and skip unneeded data
- Split file into partitions for parallel read

Data layout

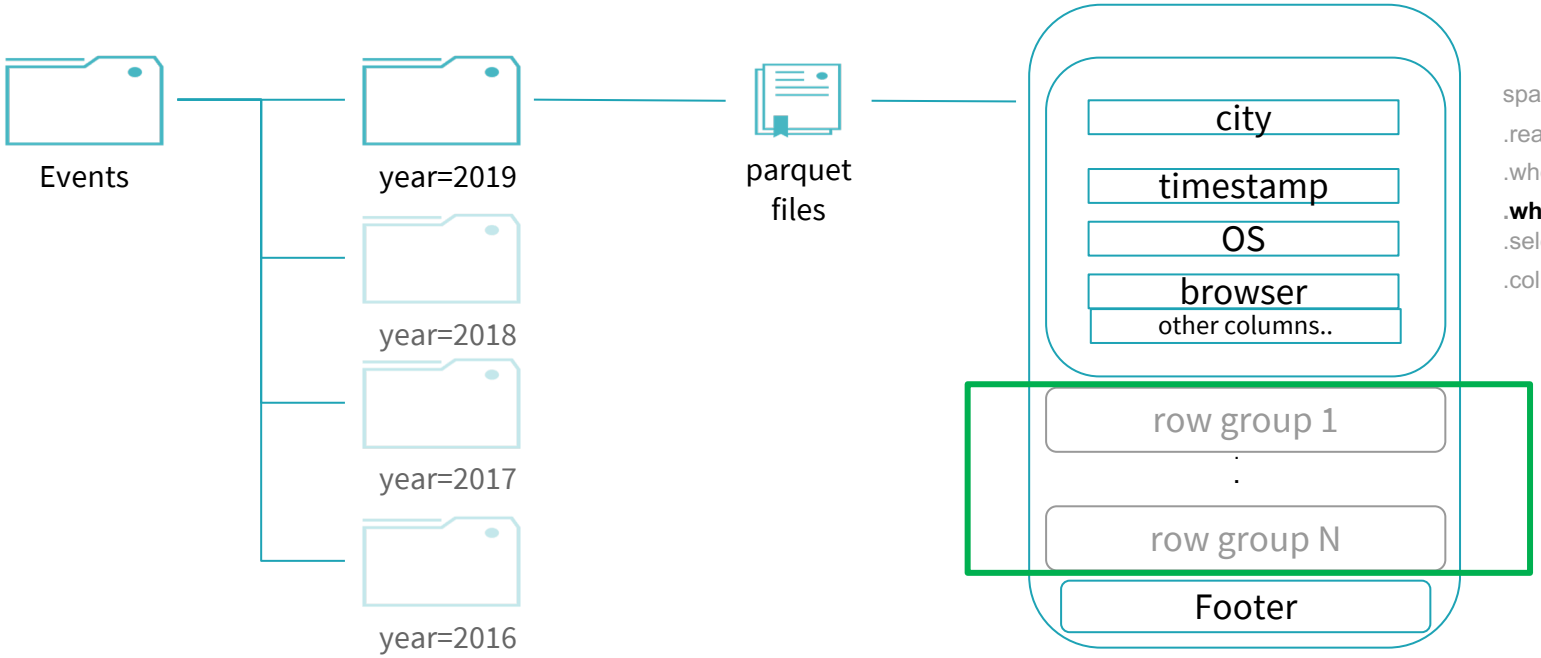

Events
year=2019
year=2018
year=2017
year=2016
parquet files
city
timestamp
OS
browser
other columns..
row group 1
row group N
Footer

Partition pruning

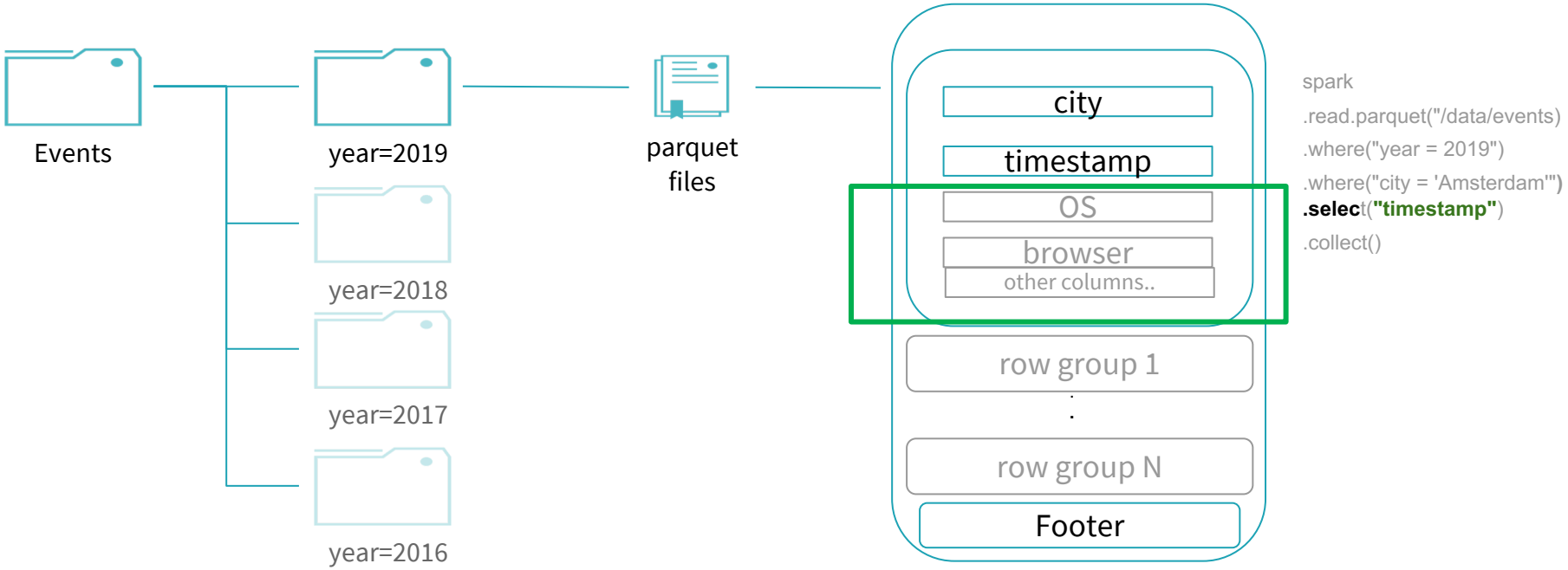



```
spark
.read.parquet("/data/events)
.where("year = 2019")
.where("city = 'Amsterdam'")
.select("timestamp")
.collect()
```

Row group skipping



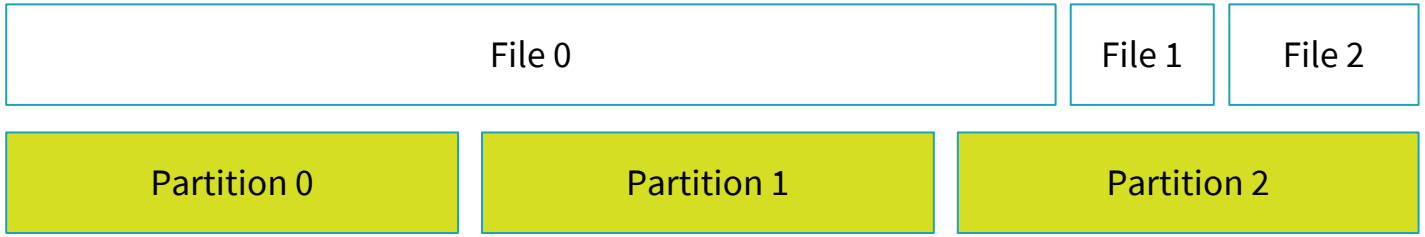
Column pruning



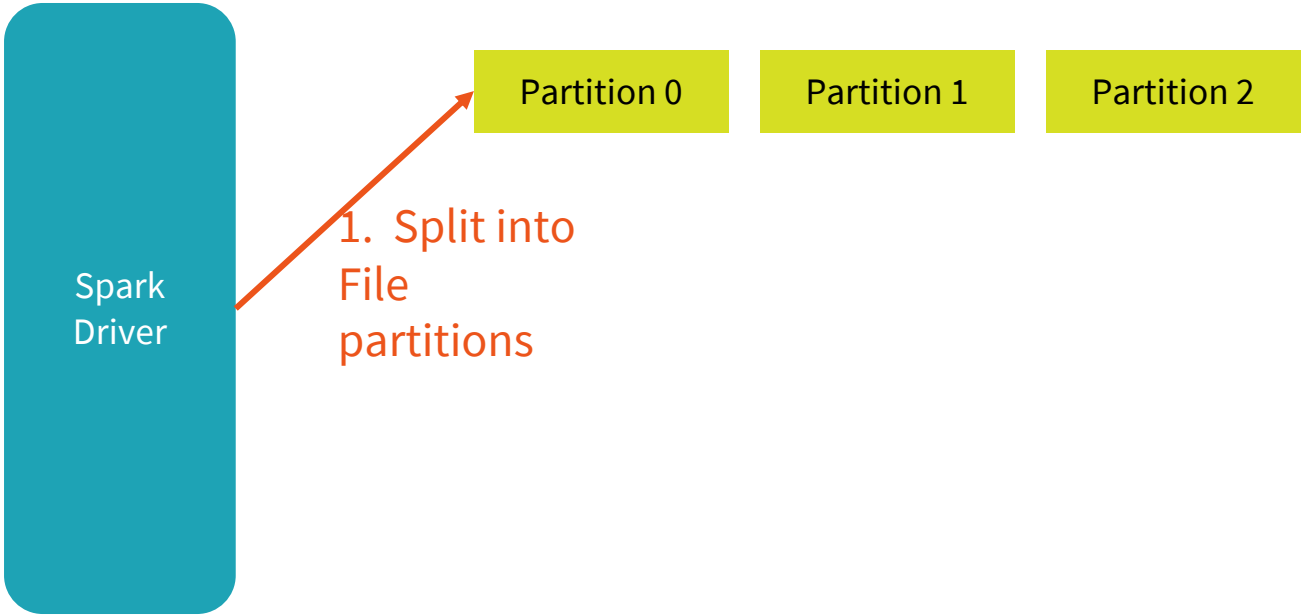
Goal

- Understand data and skip unneeded data
- Split file into partitions for parallel read

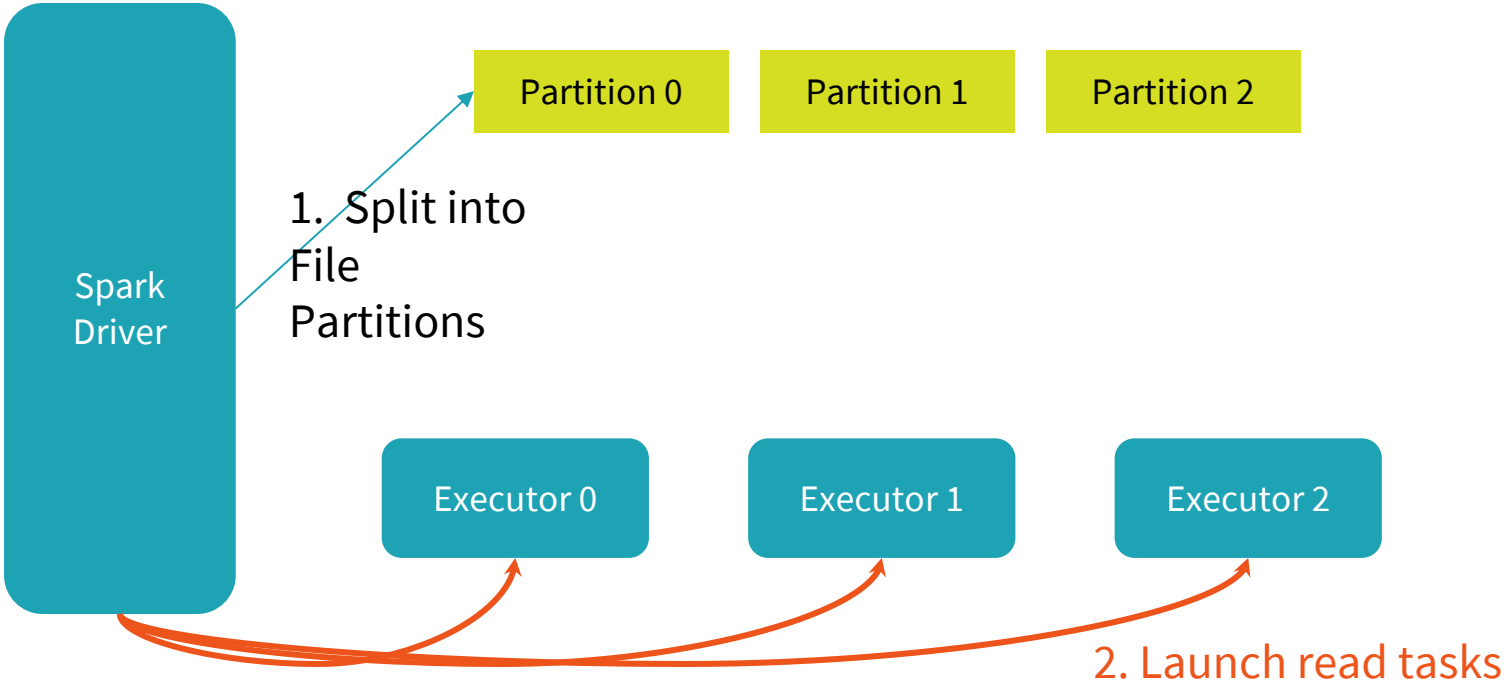
Partitions of same size



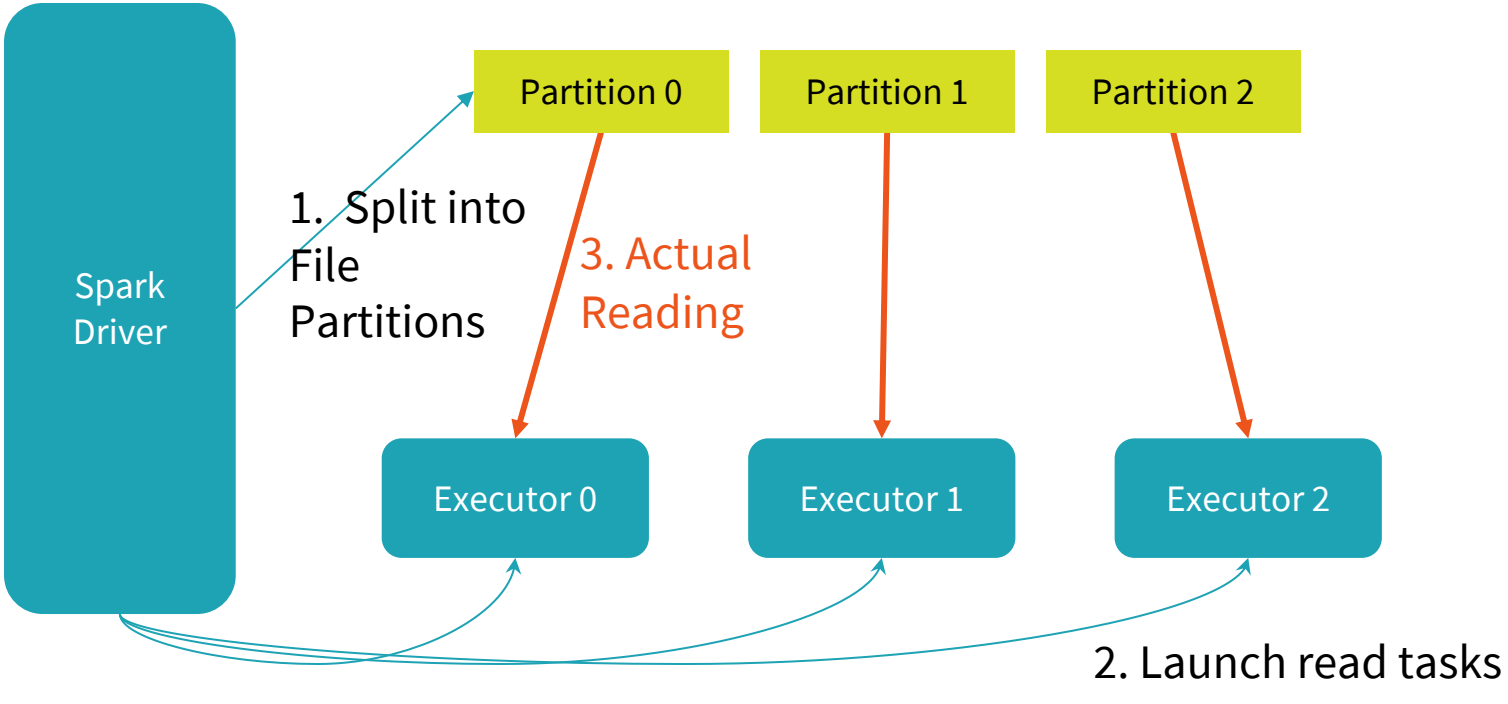
Driver: plan input partitions



Driver: plan input partitions



Executor: Read distributedly



Agenda

- File formats
- Data layout
- File reader internals
- **File writer internals**

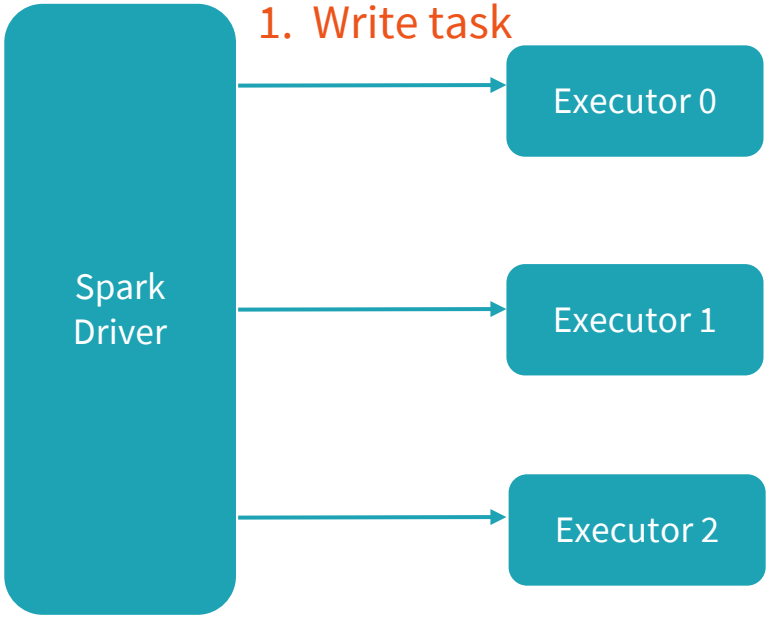
Query example

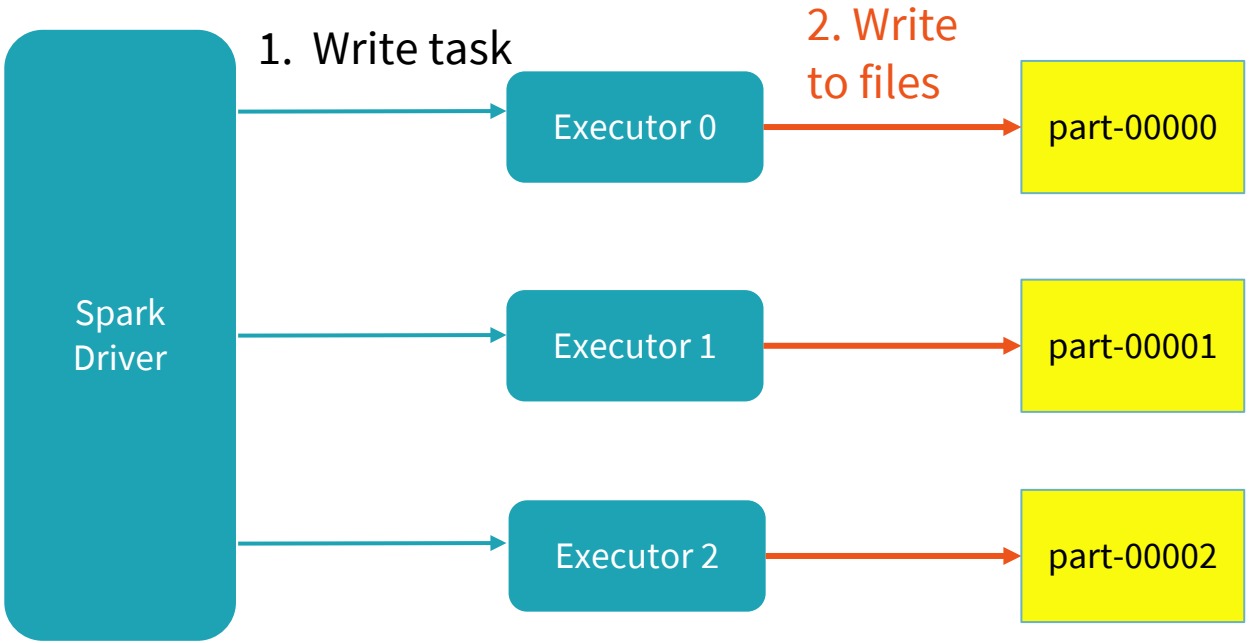
```
val trainingData =
 spark.read.parquet("/data/events")
  .where("year = 2019")
  .where("city = 'Amsterdam'")
  .select("timestamp")

trainingData.write.parquet("/data/results")
```

Goal

- Parallel
- Transactional

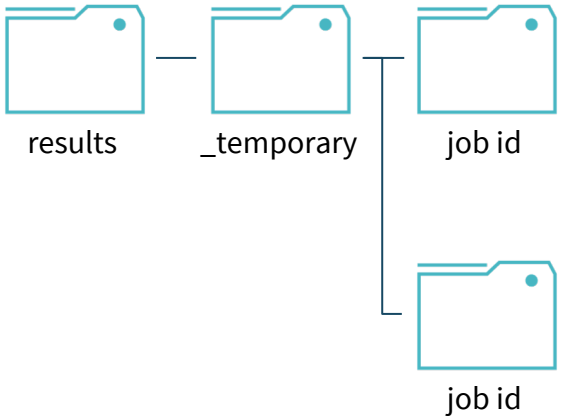

1. Write task
Spark Driver
Executor 0
Executor 1
Executor 2



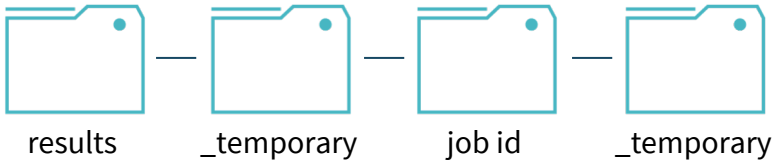
Each task writes to different temporary paths

Everything should be temporary

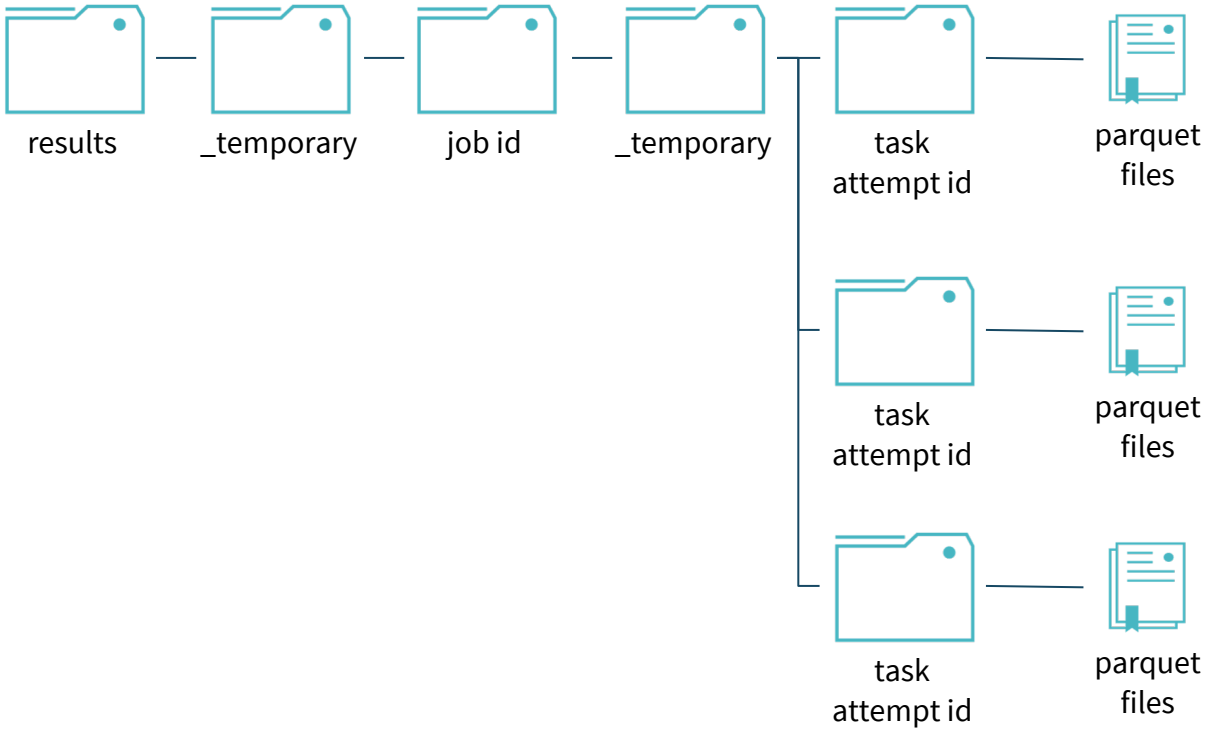
Files should be isolated between jobs



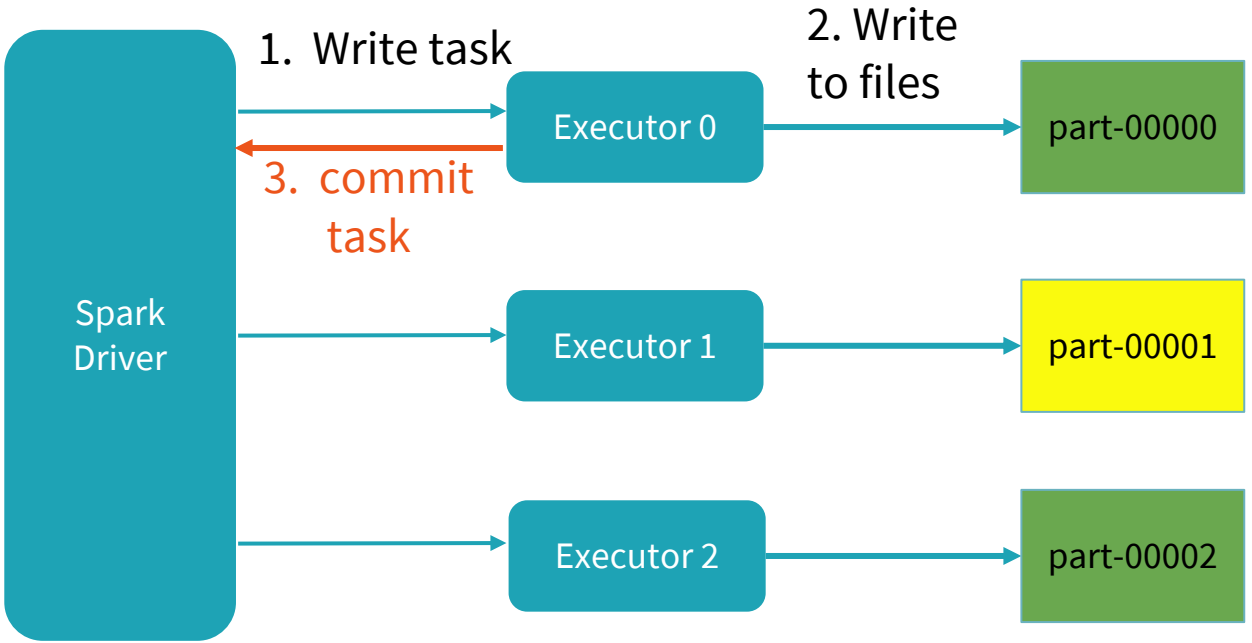
Task output is also temporary



Files should be isolated between tasks

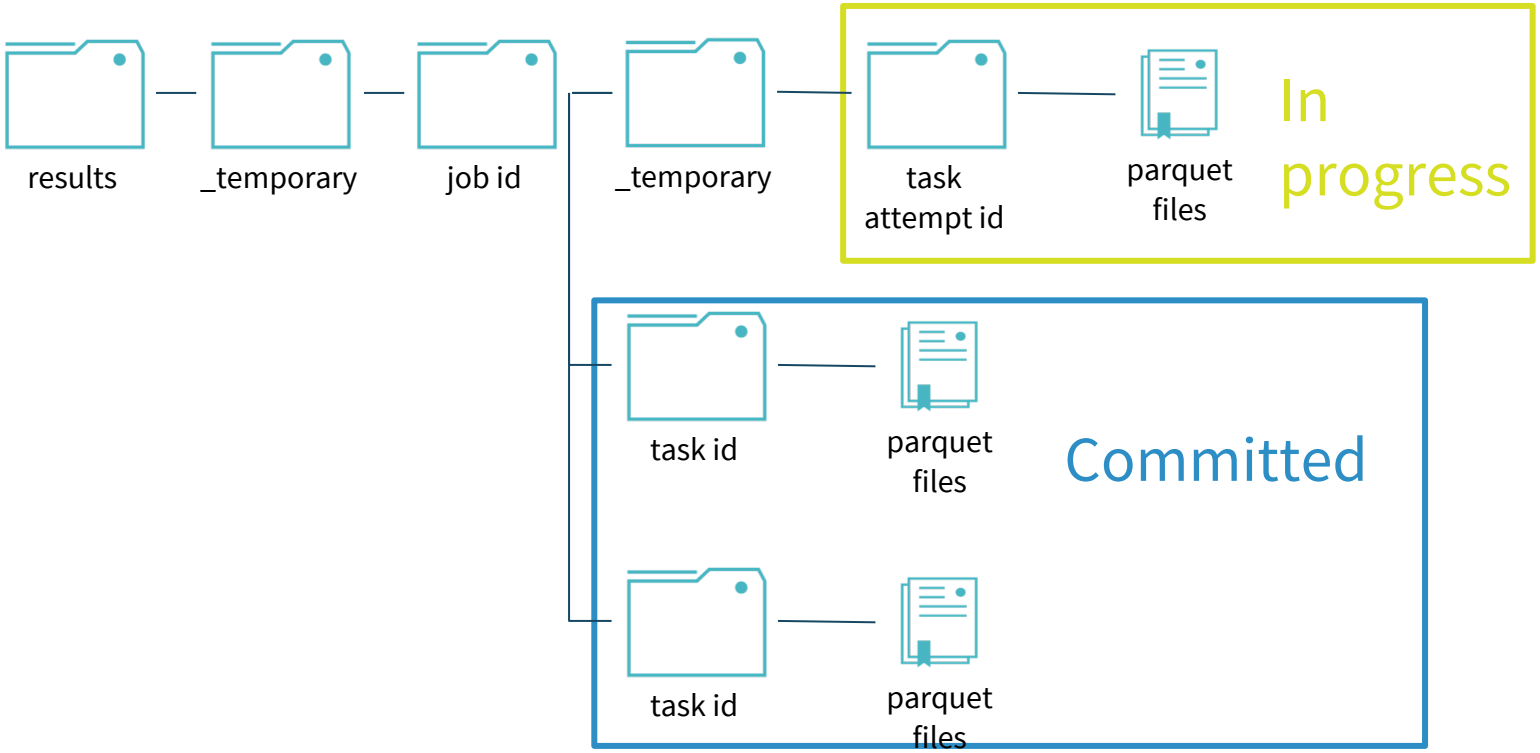


Commit task

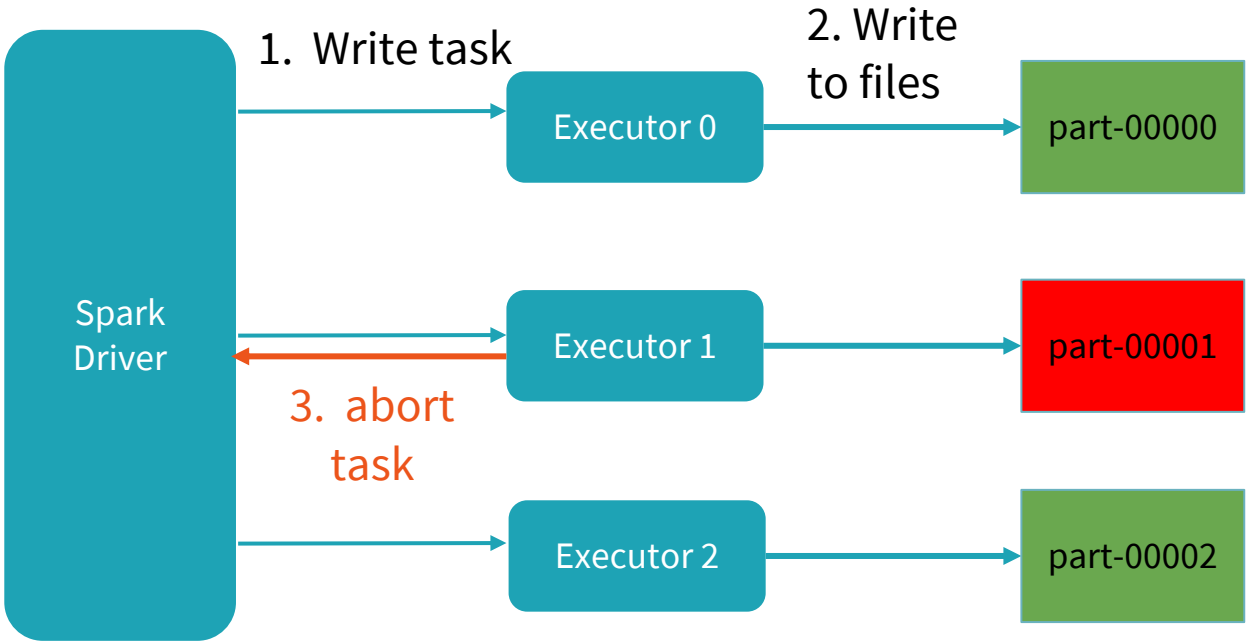


Each task writes to different temporary paths

File layout

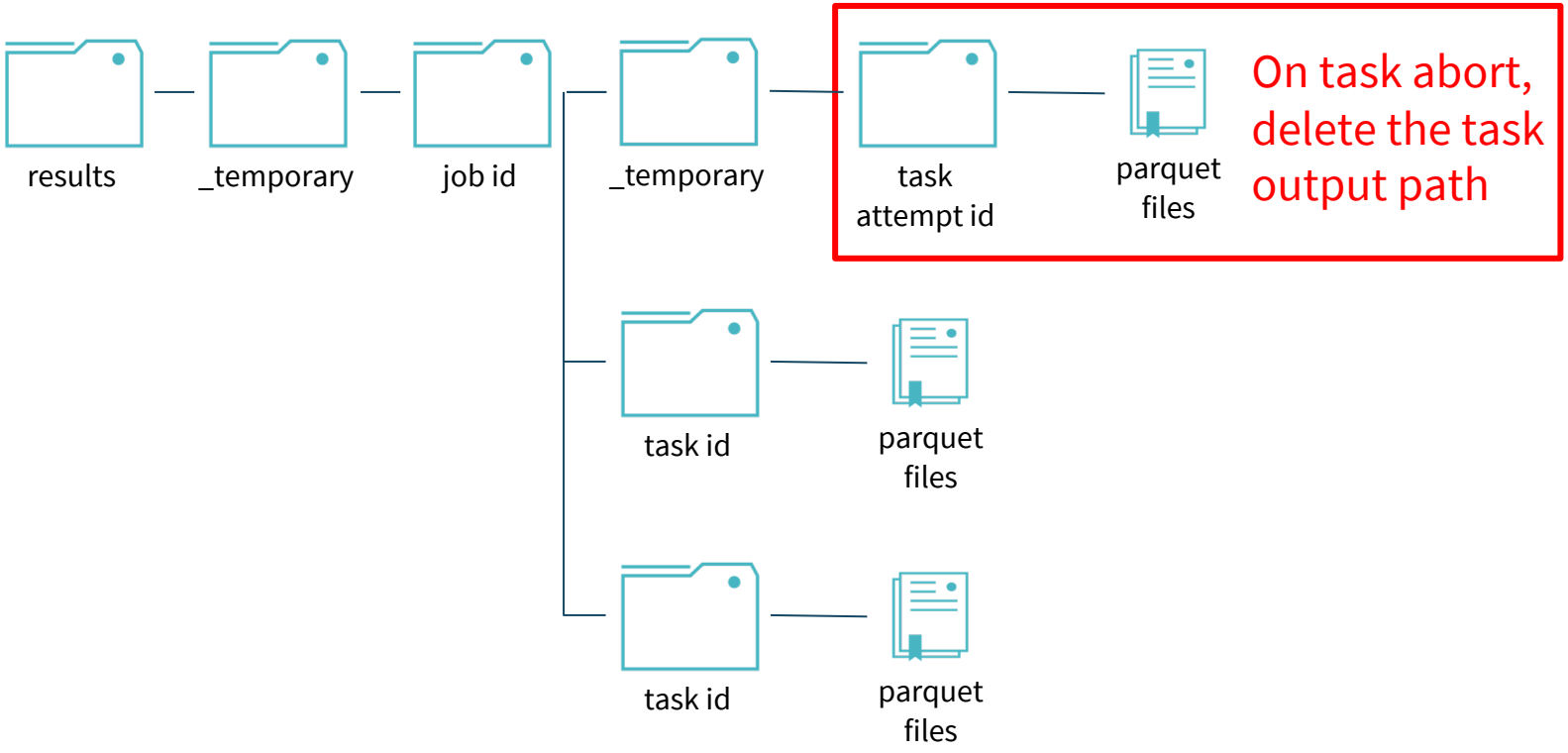

results
_temporary
job id
_temporary
task attempt id
parquet files
In progress
task id
parquet files
Committed
task id
parquet files

If task aborts..

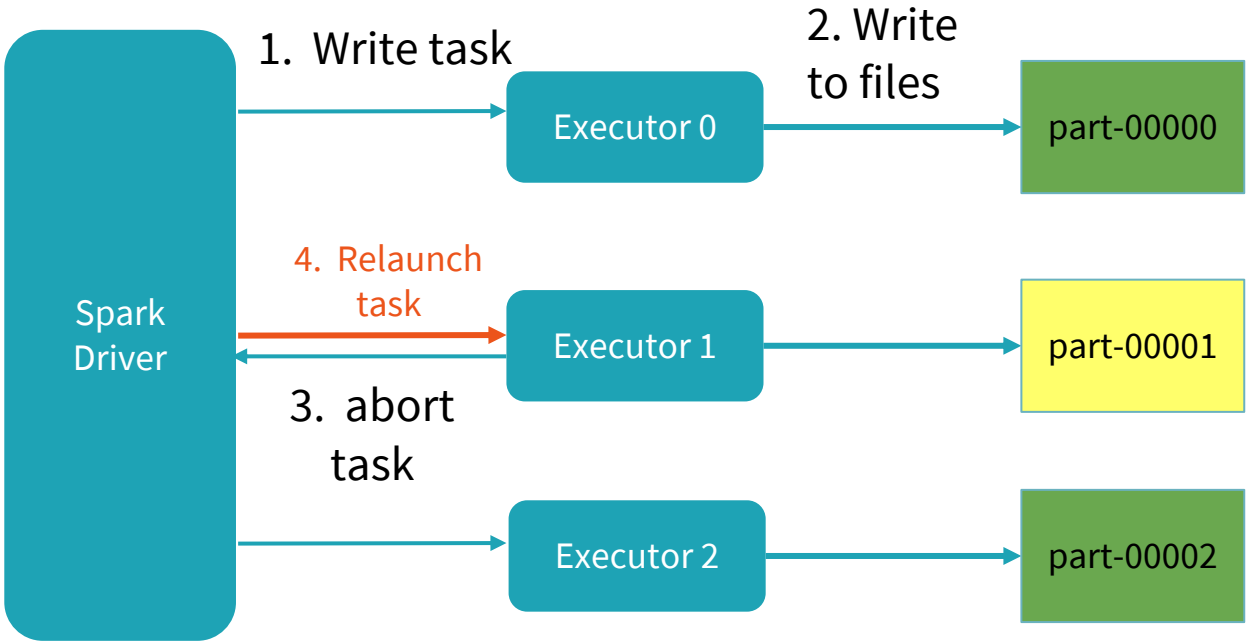


Each task writes to different temporary paths

File layout

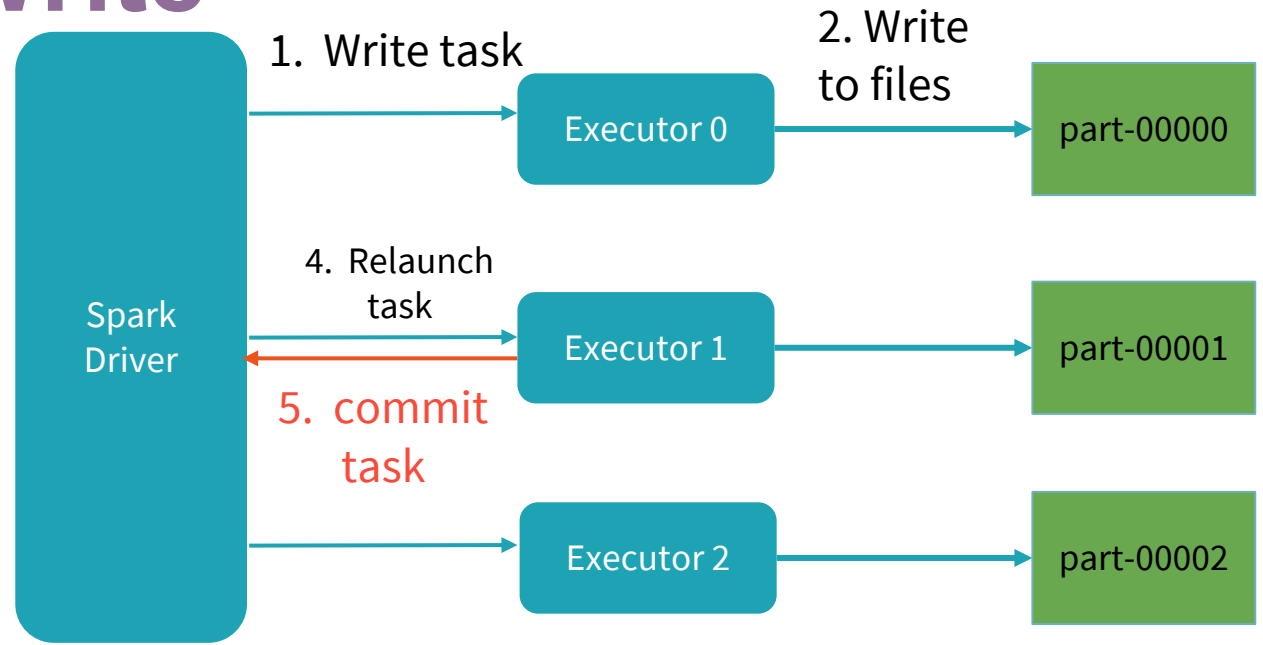

results
_temporary
job id
_temporary
task attempt id
parquet files
On task abort, delete the task output path
task id
parquet files
task id
parquet files

If task aborts..



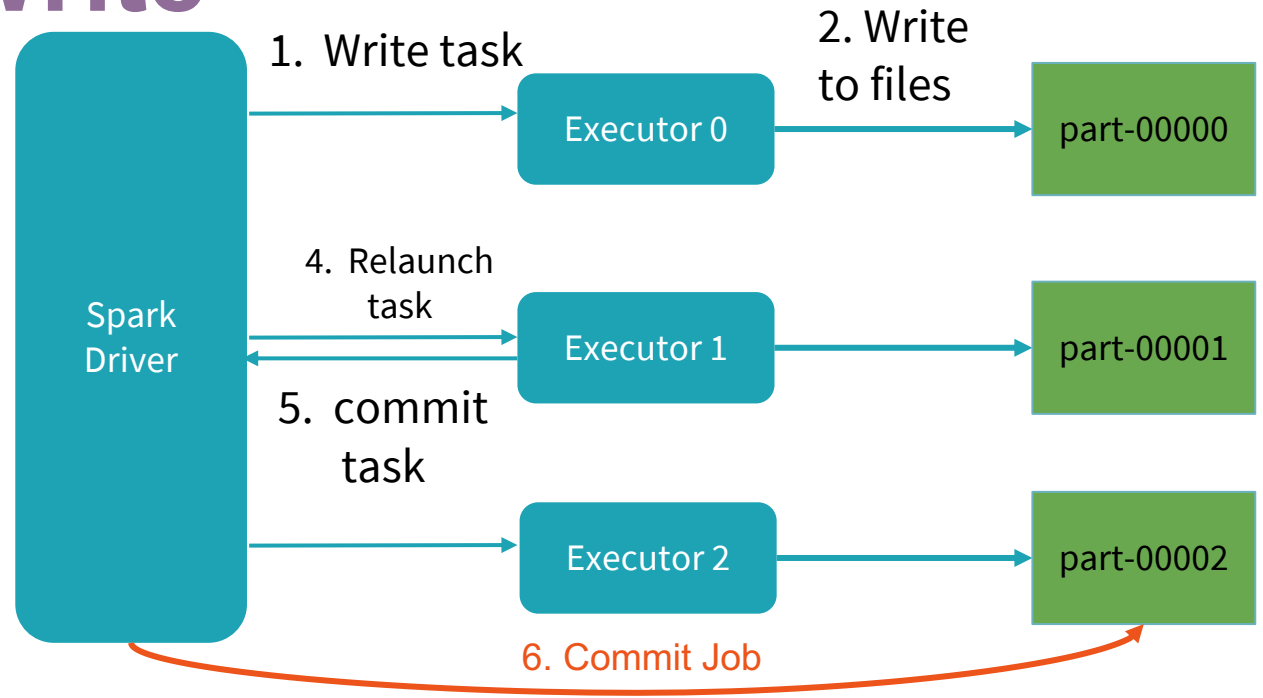
Each task writes to different temporary paths

Distributed and Transactional Write



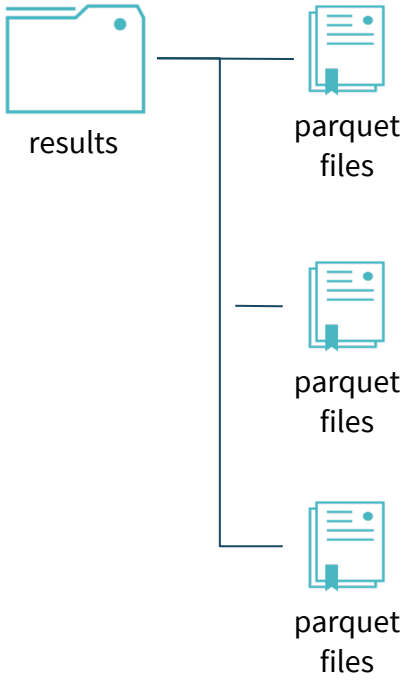
Each task writes to different temporary paths

Distributed and Transactional Write

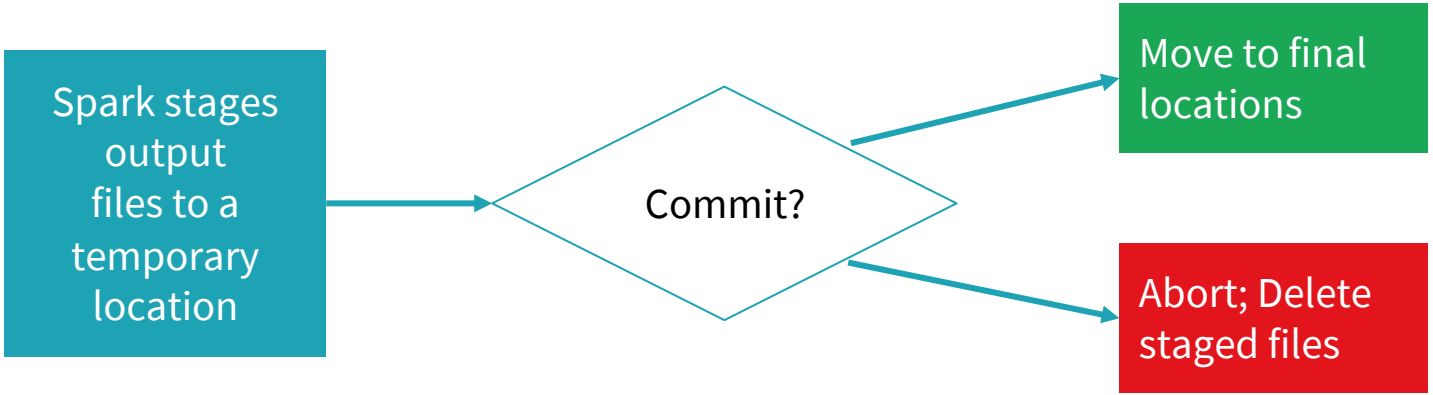


Each task writes to different temporary paths

File layout



Almost transactional



See more for concurrent reads and writes in cloud storage

Transactional writes to cloud storage – Eric Liang

Diving Into Delta Lake: Unpacking The Transaction Log – Databricks blog

Recap

File formats

- Column-oriented
  - Parquet
  - ORC
- Row-oriented
  - Avro
  - JSON
  - CSV
  - Text
  - Binary

Data layout

- Partitioning
- Bucketing

File reader

- Understand data and skip unneeded data
- Split file into partitions for parallel read

File writer internals

- Parallel
- transactional


SPARK+AI
SUMMIT 2019
Thank you!
Q & A

SPARK+AI SUMMIT 2019

DON'T FORGET TO RATE AND REVIEW THE SESSIONS

SEARCH SPARK + AI SUMMIT

Download on the App Store

GET IT ON Google Play