DATA+AI SUMMIT 2022

Recent Parquet Improvements in Apache Spark

Chao Sun

Software Engineer, Apple

ORGANIZED BY databricks

Short Intro

- Software Engineer at Apple
- Working on Spark, Hadoop, Parquet, Iceberg, Arrow and related technologies
  - Mostly focusing on improving Spark SQL performance
- Committer to Spark, Hadoop, Hive and Arrow

Motivation

- Parquet is a very popular file format, used by Spark and many other projects
- File scan is an expensive operation within a typical Spark query
- Therefore, if we can improve scan performance, we are able to reduce query end-to-end time and improve its efficiency, saving **$$$**

Outline

- Short introduction on Apache Parquet
- Complex type support for vectorized Parquet reader
- Parquet column index support in Spark
- Future work

Outline

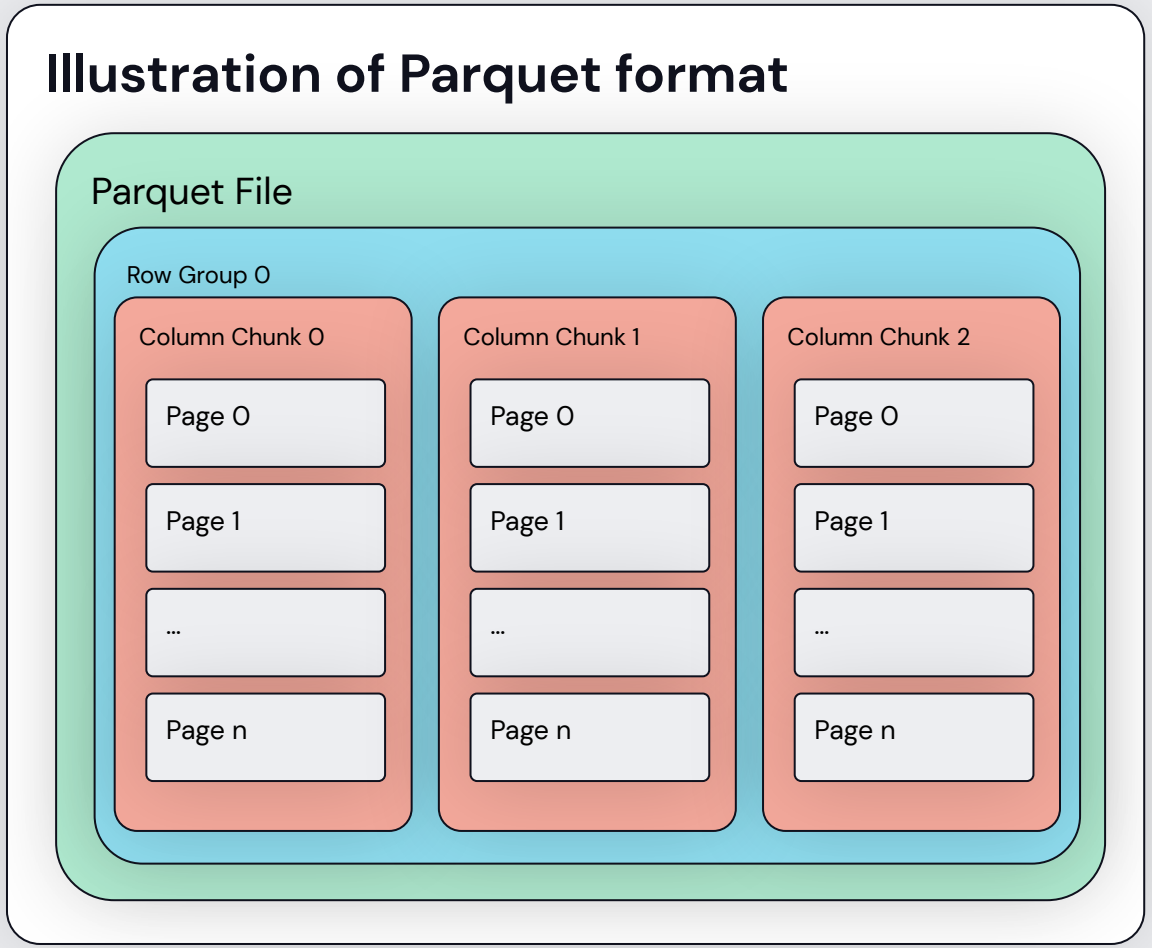
- Short introduction on Apache Parquet
- Complex type support for vectorized Parquet reader
- Parquet column index support in Spark
- Future work

Introduction on Apache Parquet

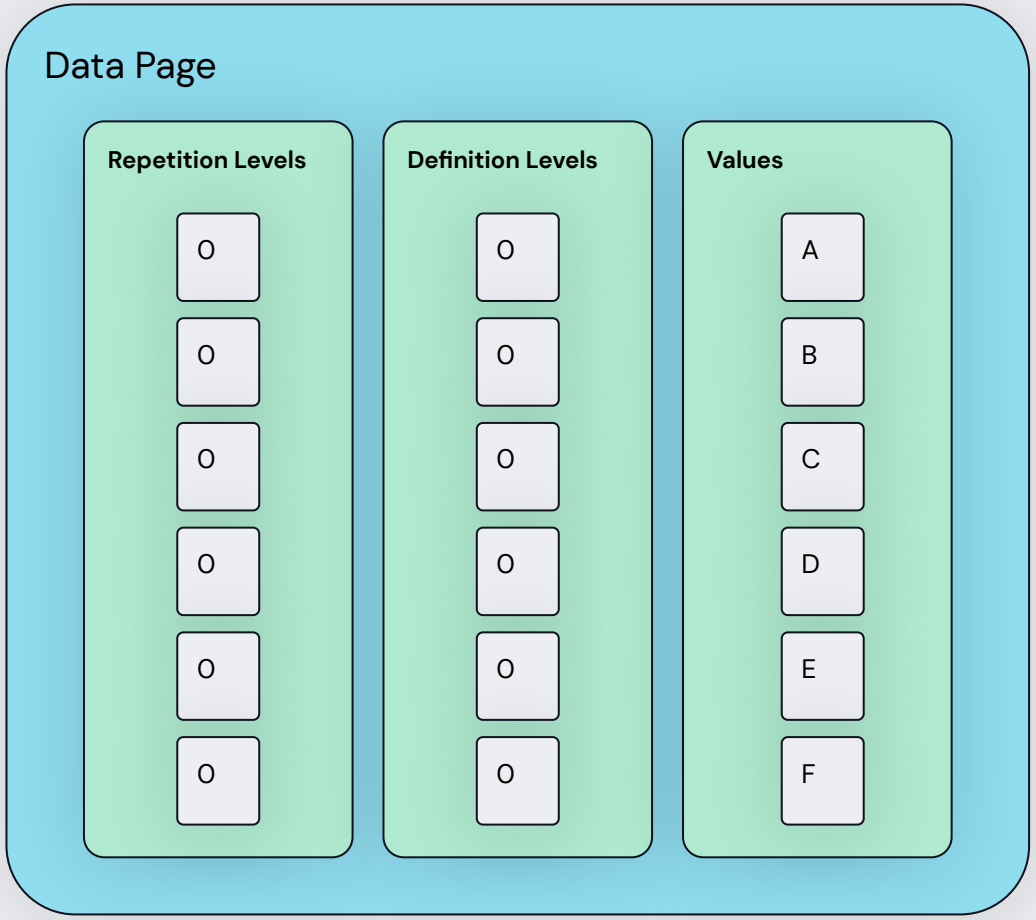
- What is Parquet:
  - A columnar format with complex types (e.g., struct, list, map) as first class citizens
  - Inspired by the Dremel paper from Google
- A single format specification, with different implementations
  - In different projects: Spark, Trino/Presto, Iceberg, Impala, Hive, etc
  - In different languages: Java, C++, Rust, Go, etc
- Widely used in the industry

Parquet: Glossary

- Row Group
  - Consists of one or more **column chunks**, one for each column in the file schema
  - **128mb** by default
- Column Chunk
  - A chunk of data for a column
  - Consists of one or more **pages**
  - Also contains statistics for the column
- Page
  - Basic unit for compression and encoding
  - 2 types of pages: *dictionary* and *data*
  - **1mb** by default



Parquet: Data Page

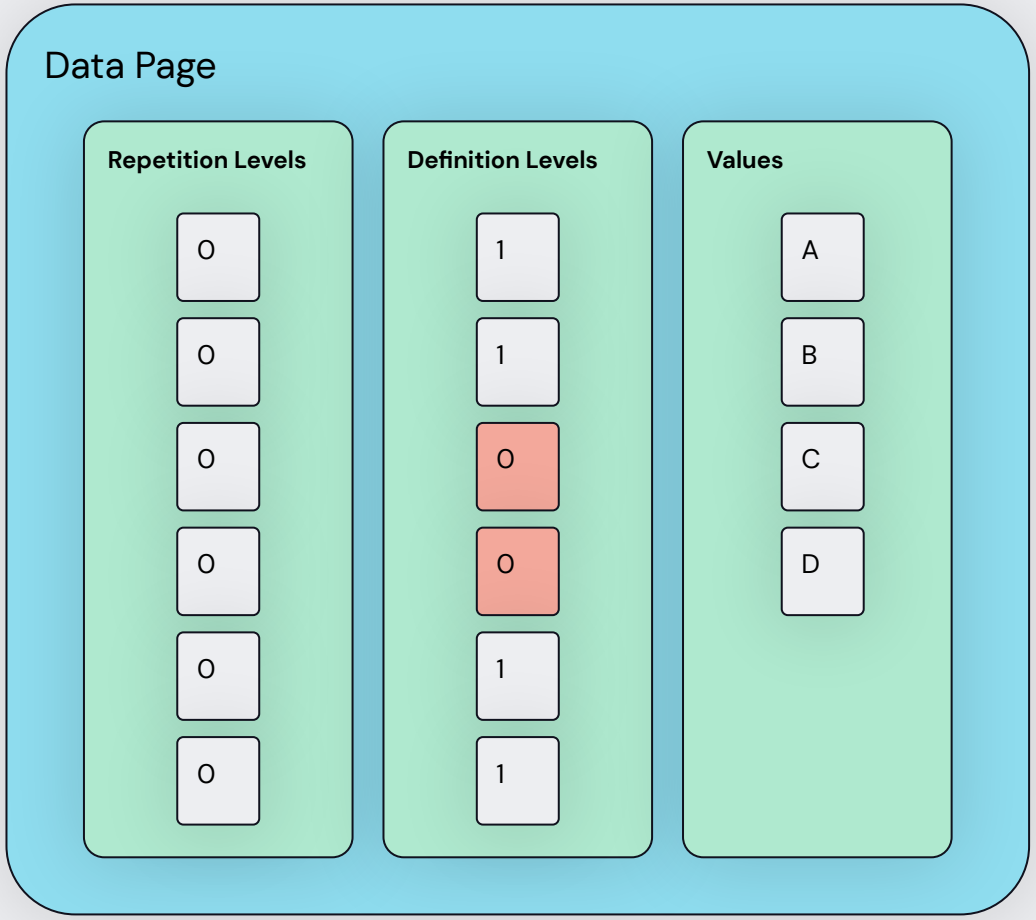


Repetition levels
- Represent the start of a new record

Definition levels
- Represent null-ness of values

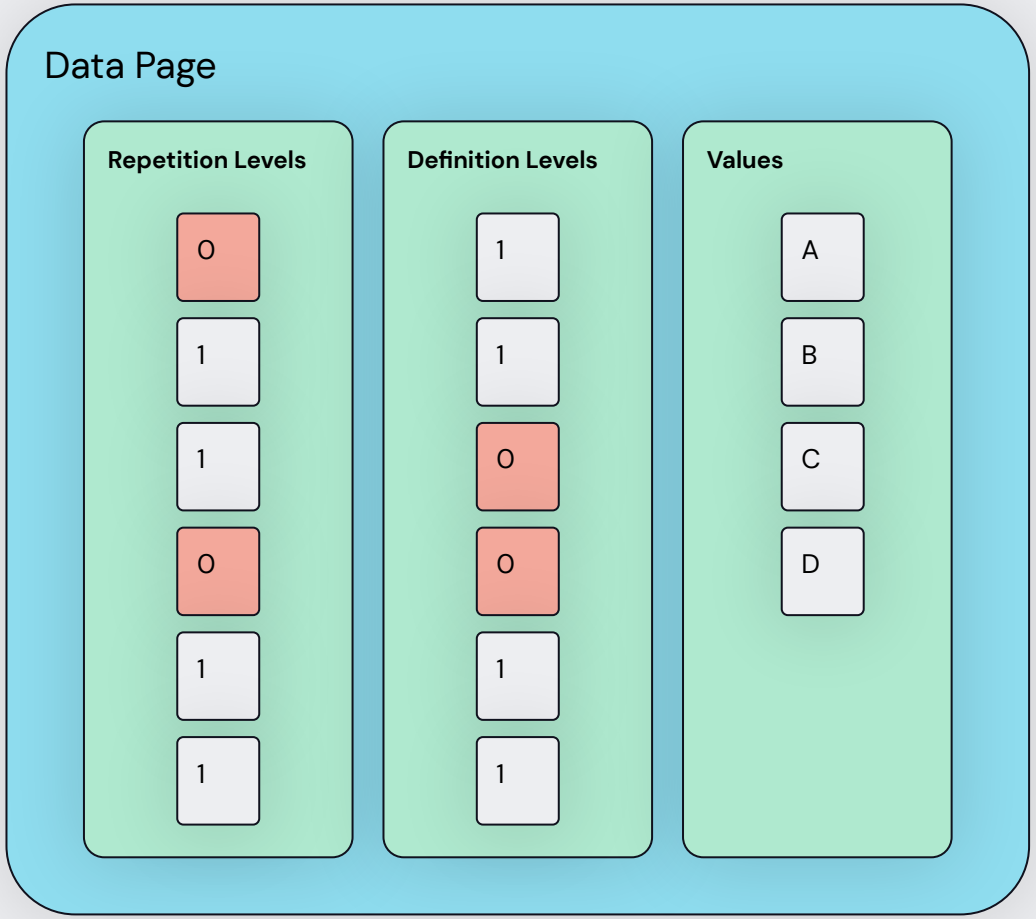
Values
- Actual non-null values of data

Parquet: Data Page



- Column type: **`string`**
- Represents: **`[A, B, null, null, C, D]`**
- For this particular schema:
  - Definition level == 1 means the value is not-null, while 0 means the value is null
- The 3rd and 4th elements are null, since the corresponding definition level is 0

Parquet: Data Page



- Column type: **`list<string>`**
- Represents: **`[[A, B, null], [null, C, D]]`**
- For this particular schema:
- Definition level == 1 means the value is not-null, while 0 means the value is null
- The 3rd and 4th elements are null, since the corresponding definition level is 0
- **Repetition level == 1** indicates the start of a new list

Outline

- Short introduction on Apache Parquet
- Complex type support for vectorized Parquet reader
- Parquet column index support in Spark
- Future work

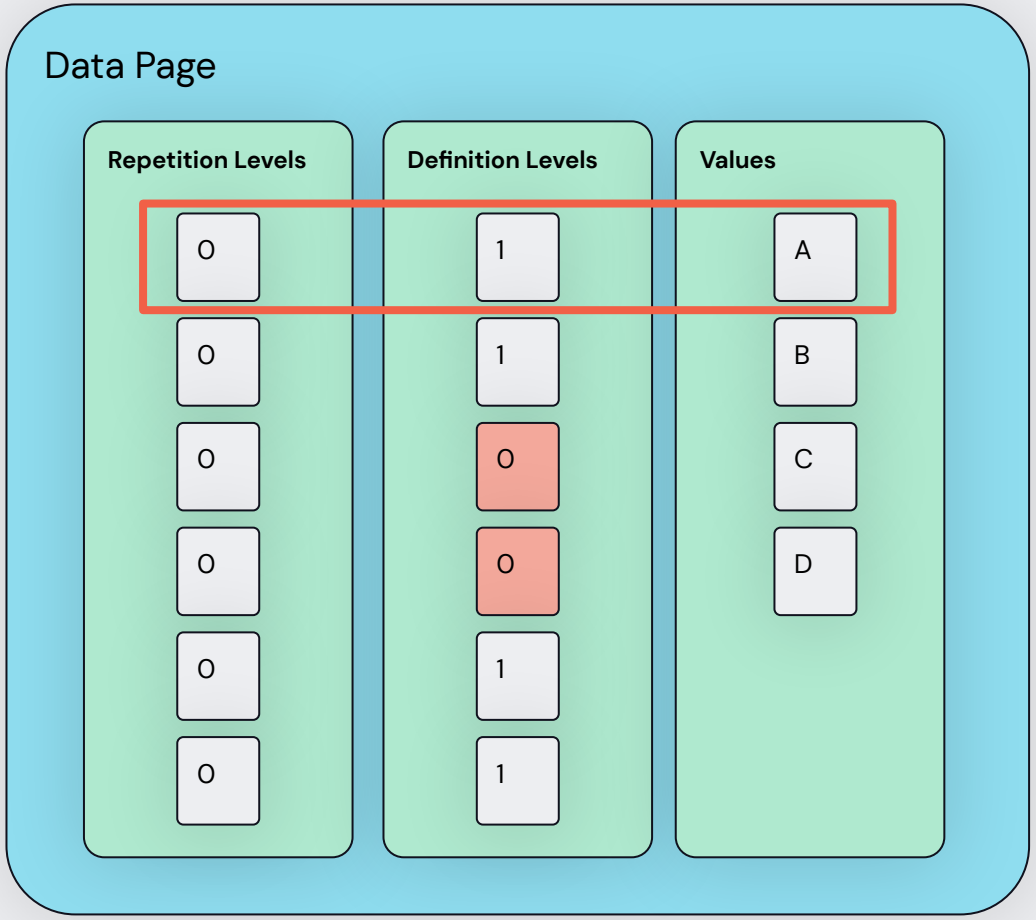
Background

- Complex Type:
  - Struct, e.g: `struct<f1: int, f2: string>`
  - List, e.g.: `list<string>`
  - Map, e.g: `map<int, string>`

Two types of Parquet readers in Spark

- Non-vectorized reader (fallback)
  - Uses reader implementation from Parquet Java project (aka **parquet-mr**)
  - Support all types (including complex types)
- Vectorized reader (default)
  - Re-written from scratch in Spark
  - Support primitive types (e.g., int/float/string/decimal/timestamp/etc)
  - Scan data in batches (hence called *vectorized*)

Non-Vectorized Parquet Reader

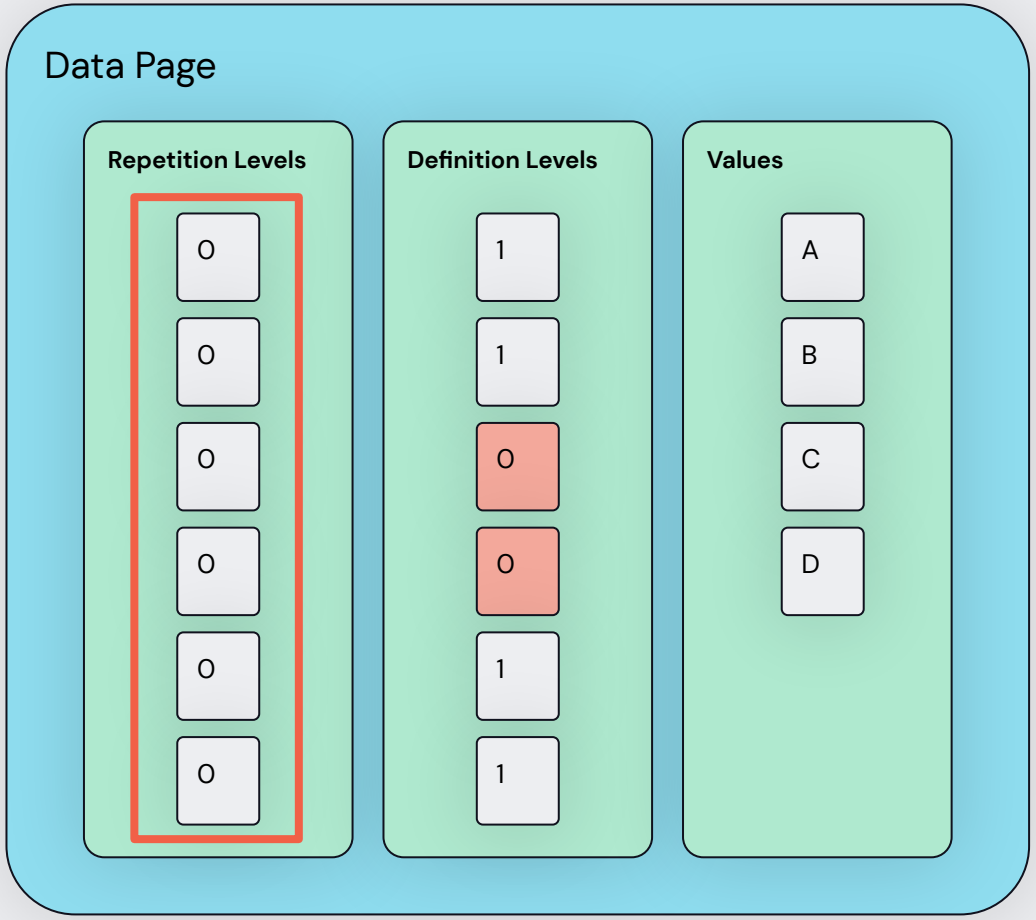



```
while (scan.hasNext()) {
  val row = scan.next()
  // compute
}
```

Steps (for each column)

1. Read the next repetition level
2. Read the next definition level
3. If value is not null, read the next value
4. Assemble into Spark record and pass to computation (e.g., filter, join, aggregation, sort)

Vectorized Parquet Reader




```
while (scan.hasNext()) {
  val batch = scan.next()
  // compute
}
```

Steps (for each column)

1. Read the next batch of repetition level
2. Read the next batch of definition level
3. If value is not null, read the next batch of values
4. Assemble into columnar batch and pass to computation

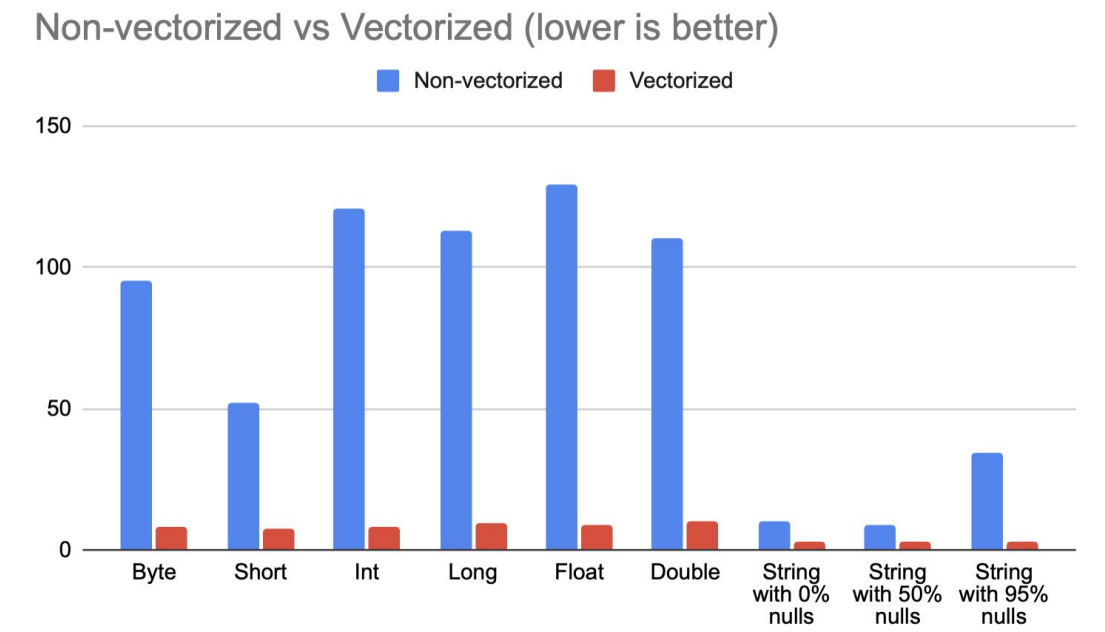
Advantages of Vectorized Approach

- Much better memory locality and cache utilization
- Uses `memcpy` when reading batches of values
- Encoding specific optimizations

Perf: vectorized vs non-vectorized

- Between 10-20x improvements for primitive types
- Improvements are more significant for string type when there is high cardinality of nulls

Micro benchmark result



High Level Idea

- Annotation maximum repetition and definition level when converting Parquet schema to Spark schema
  - Also need to handle legacy formats for list and map
- Read & materialize repetition levels, definition levels and values.
  - Optimization: if repetition or definition levels are not needed, materialization is skipped
- Assemble columnar batch recursively, starting from leaf schema nodes

Parquet Schema Conversion

Parquet: `list<list<int32>>`

```
optional group array_of_arrays (LIST) {
  repeated group list {
    required group element (LIST) {
      repeated group list {
        optional int32 element;
      }
    }
  }
}
```

Parquet: `list<list<int32>>`

```
optional group array_of_arrays (LIST) {
  repeated group list {
    required group element (LIST) {
      repeated group list {
        optional int32 element;
      }
    }
  }
}
```

R=0 D=1
R=1 D=2
R=1 D=2
R=2 D=3
R=2 D=4

Spark: `ArrayType<ArrayType<IntegerType>>`

```
ArrayType(R=0, D=0,
  ArrayType(R=1, D=2,
    IntegerType(R=2, D=4)
  )
)
```

SPARK-34863: Complex type support

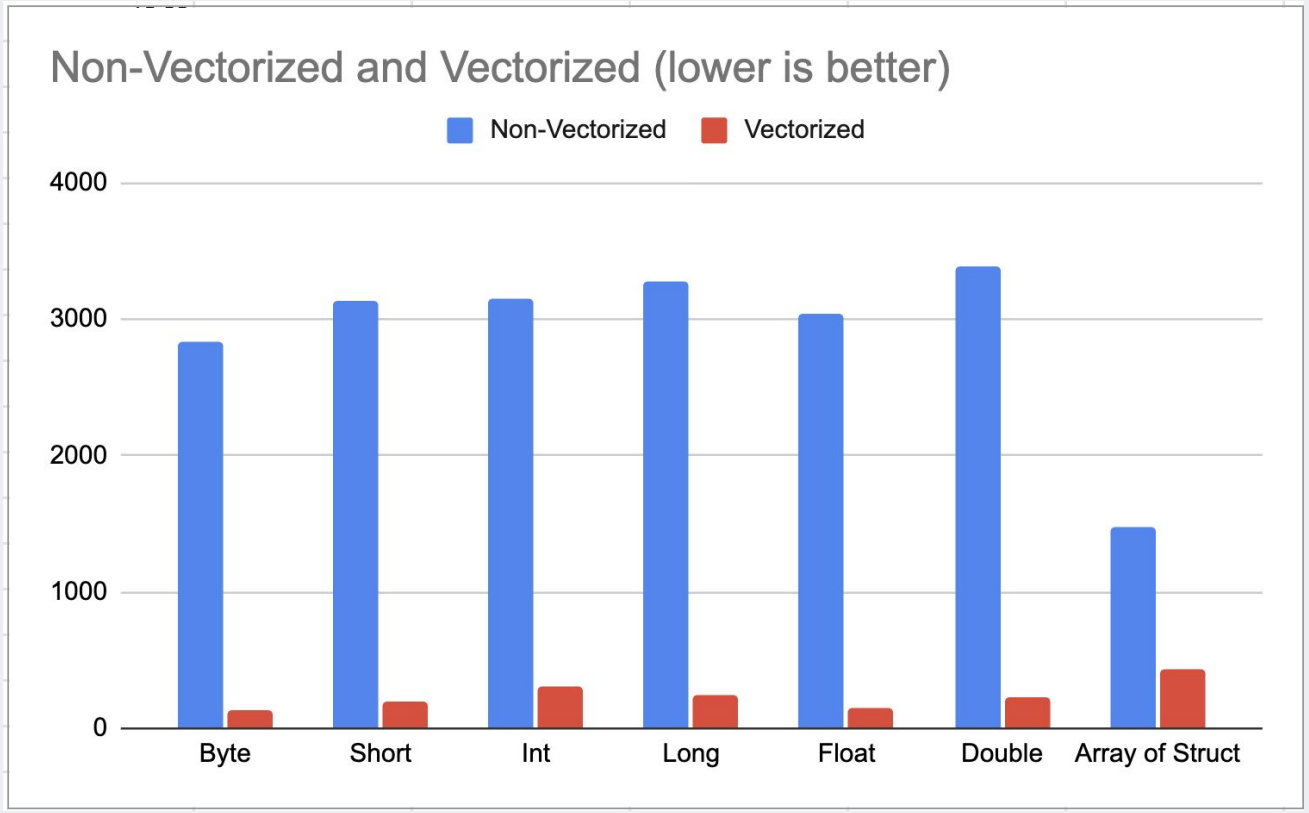
- Added support for reading Parquet data of complex types, e.g., list, map, struct.
- Added a config `spark.sql.parquet.enableNestedColumnVectorizedReader` to turn on or off the feature
  - Turned off by default in Spark 3.3
- Shipped in Spark 3.3

Complex Type – Performance

- 10-20x improvements when reading struct fields

```
SELECT s.f FROM tbl …
```

- 3.5x improvements when reading array of structs

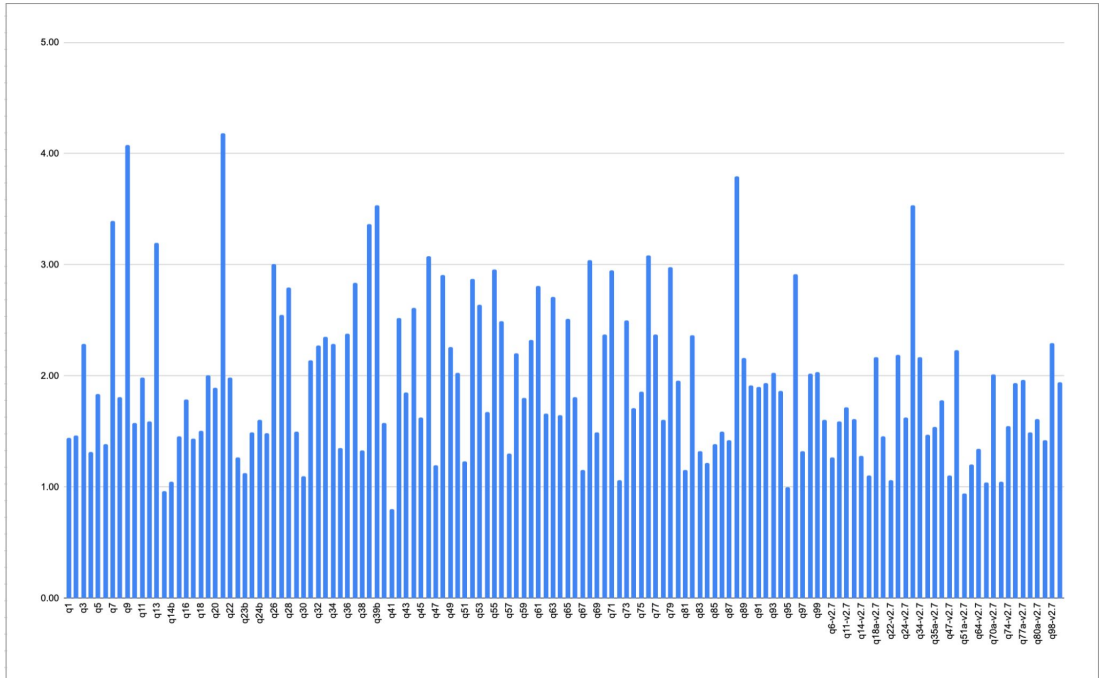


Perf: vectorized vs non-vectorized

TPC-DS (SF=1, Spark 3.1.2)

- w/ and w/o vectorization
- Average speed-up: **2x**
- Expect the same amount of improvement when reading fields from struct type, if Spark supports complex types

TPC-DS result



Outline

- Short introduction on Apache Parquet
- Complex type support for vectorized Parquet reader
- Parquet column index support in Spark
- Future work

Parquet Predicate Pushdown

Existing filter mechanisms

- Statistics
  - i.e., min/max stats
- Dictionary
  - When dictionary encoding is used, apply equality check on dictionary values
- Bloom filter (since parquet-1.12)
  - Apply equality checks on bloom filter per column chunk

All of these skip data on **row group** level

Column Index

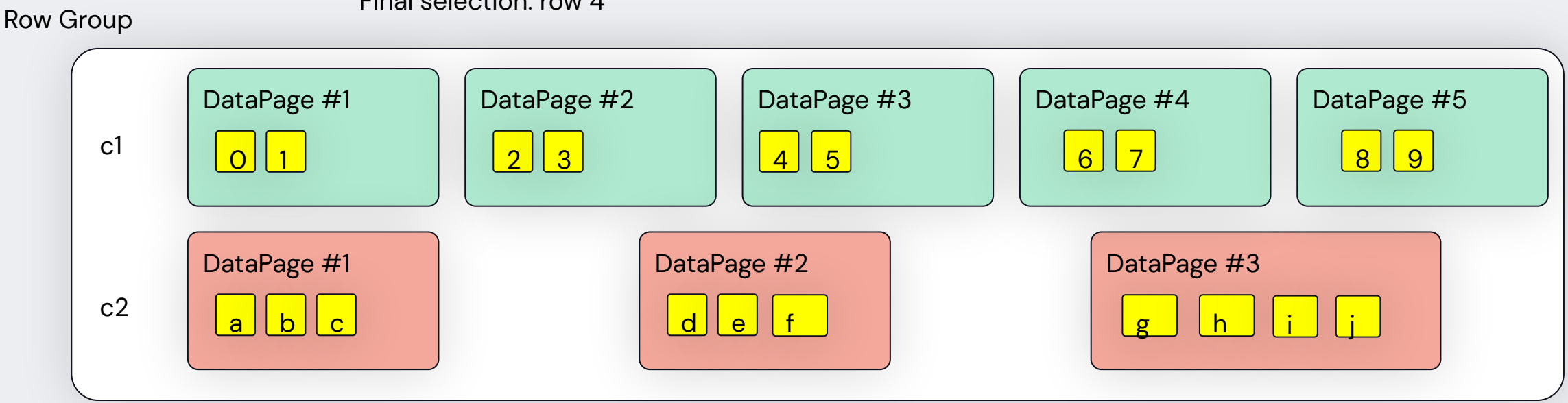
Skip data pages using **page level** min/max statistics

- Saves CPU and IO when data pages can be completely skipped
- Most effective when data is sorted, or with low selectivity filters
- Introduced in Parquet 1.11

Column Index Filtering

```
SELECT * FROM tbl WHERE c1 > 3 AND c2 < 'f'
```

c1 > 3: select rows in [4, 9]
c2 < 'f': select rows in [0, 4]
Final selection: row 4

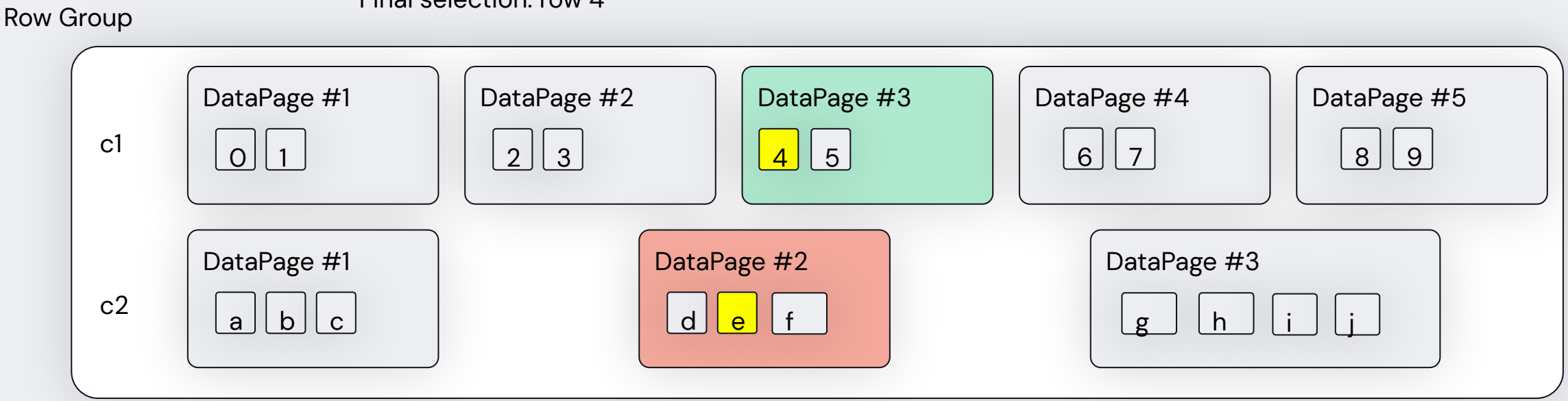


Column Index Filtering

```
SELECT * FROM tbl WHERE c1 > 3 AND c2 < 'f'
```

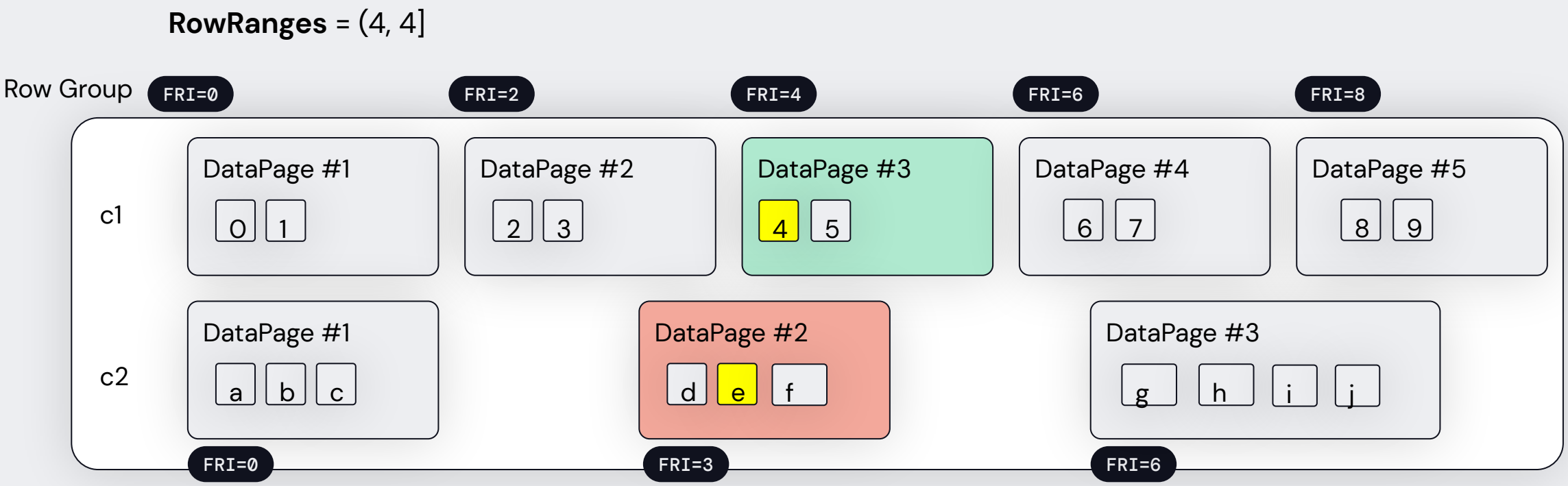
c1 > 3: select rows in [4, 9]
c2 < 'f': select rows in [0, 4]
Final selection: row 4

Other filter mechanisms have to read the entire row group



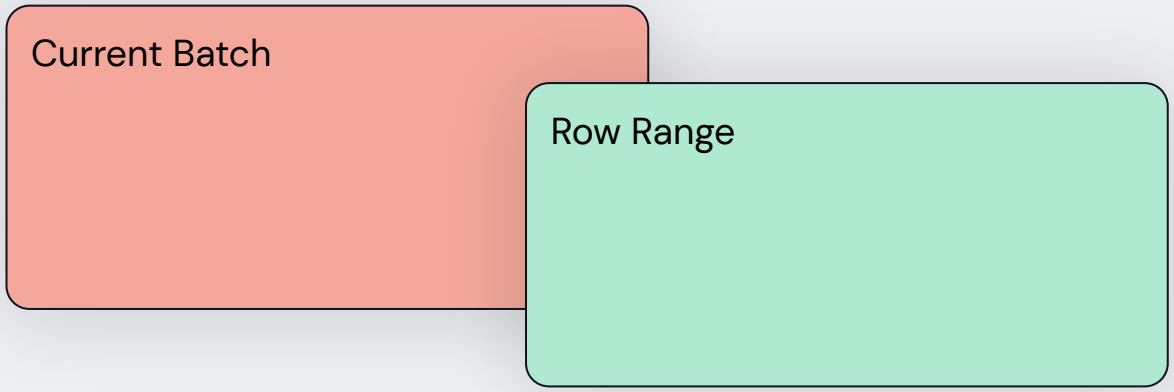
Column Index Filtering

- **FirstRowIndex (FRI)**: the first row index of a page
- **RowRange**: the range of rows that are selected



Column Index Support in Spark

- Process **FirstRowIndex** and **RowRanges** to skip Parquet records
- Add new logic to skip reading values
  - For instance, with PLAIN encoding, we can simply advance the cursor into the byte buffer by N positions
- Process RowRanges by comparing with the current batch of rows

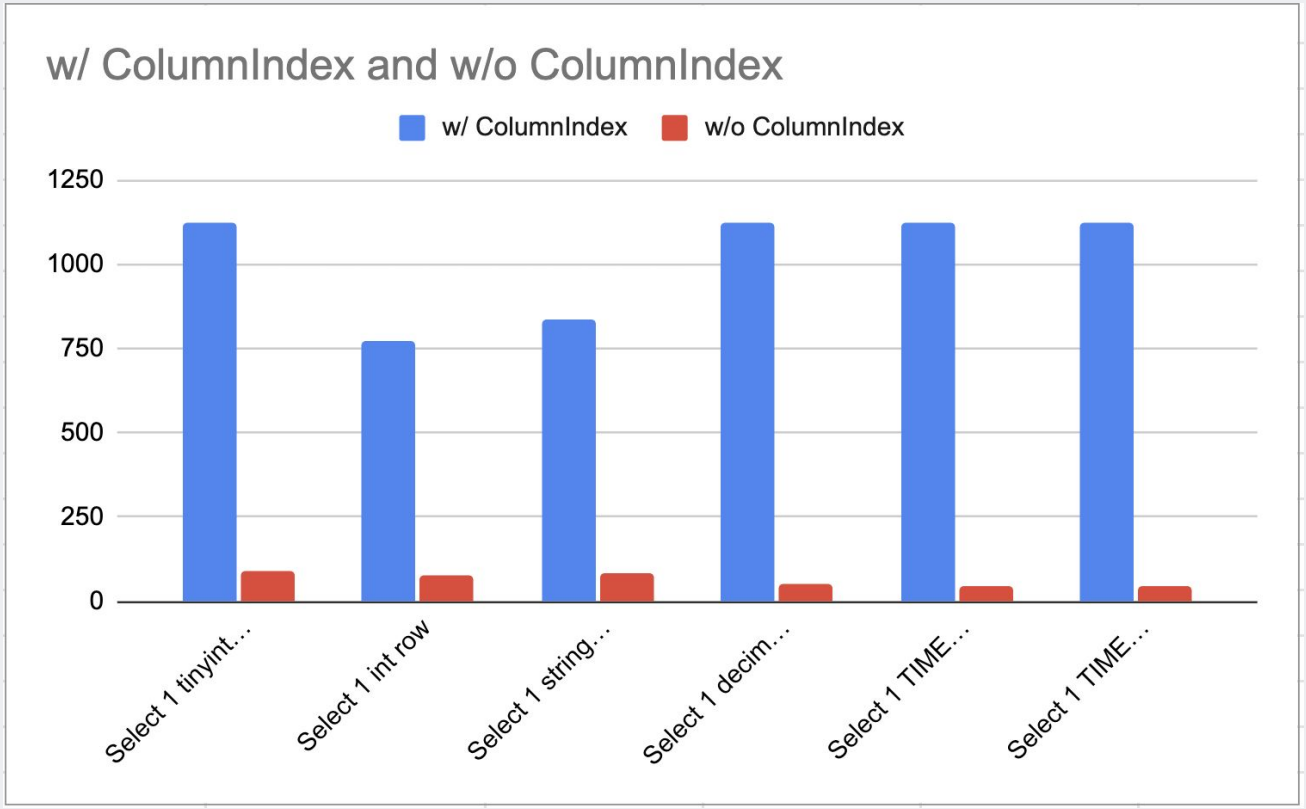


3 cases

- RowRange **before** batch
- RowRange **after** batch
- RowRange **overlap** with batch

Column Index – Performance

- Selecting a single row in 15M rows: 10–26x improvements depending on data type
- More benchmark results can be found in this [blog post]



Outline

- Short introduction on Apache Parquet
- Complex type support for vectorized Parquet reader
- Parquet column index support in Spark
- Future work

Future Work

SPARK-36529: Decoupling IO and CPU during Parquet scan

- Spark currently process Parquet row groups sequentially: first download all row group data, then decompress & decoding page by page
- In progress via PARQUET-2149 and HADOOP-11867

SPARK-36527: Lazy materialization

- Evaluate filters first, followed by materializing data
- Similar to column index, but more general
- Can apply to old Parquet files written without column index

DATA+AI
SUMMIT 2022

Thank you