SPARK+AI
SUMMIT 2019

Build. Unify. Scale.

WIFI SSID:Spark+AISummit | Password: UnifiedDataAnalytics

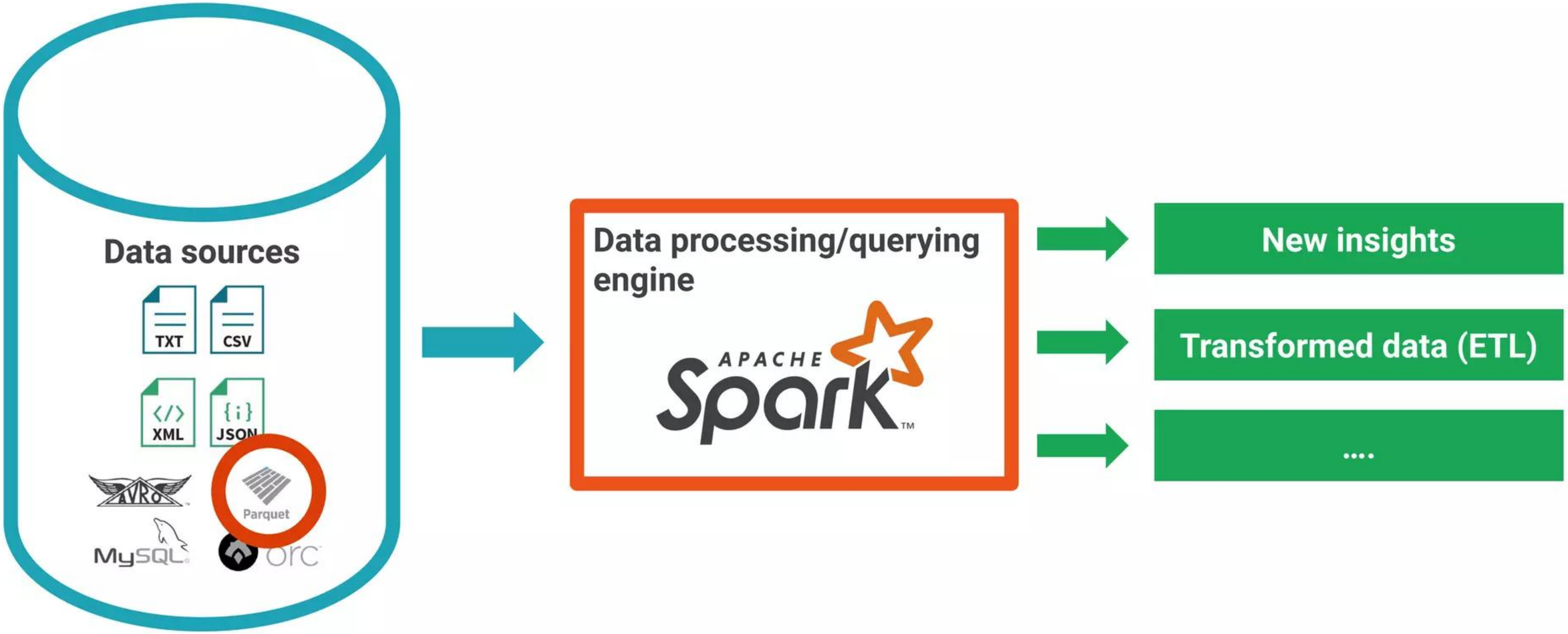
SPARK+AI SUMMIT 2019

The Parquet Format and Performance Optimization Opportunities

Boudewijn Braams
Software Engineer - Databricks

#UnifiedDataAnalytics #SparkAISummit

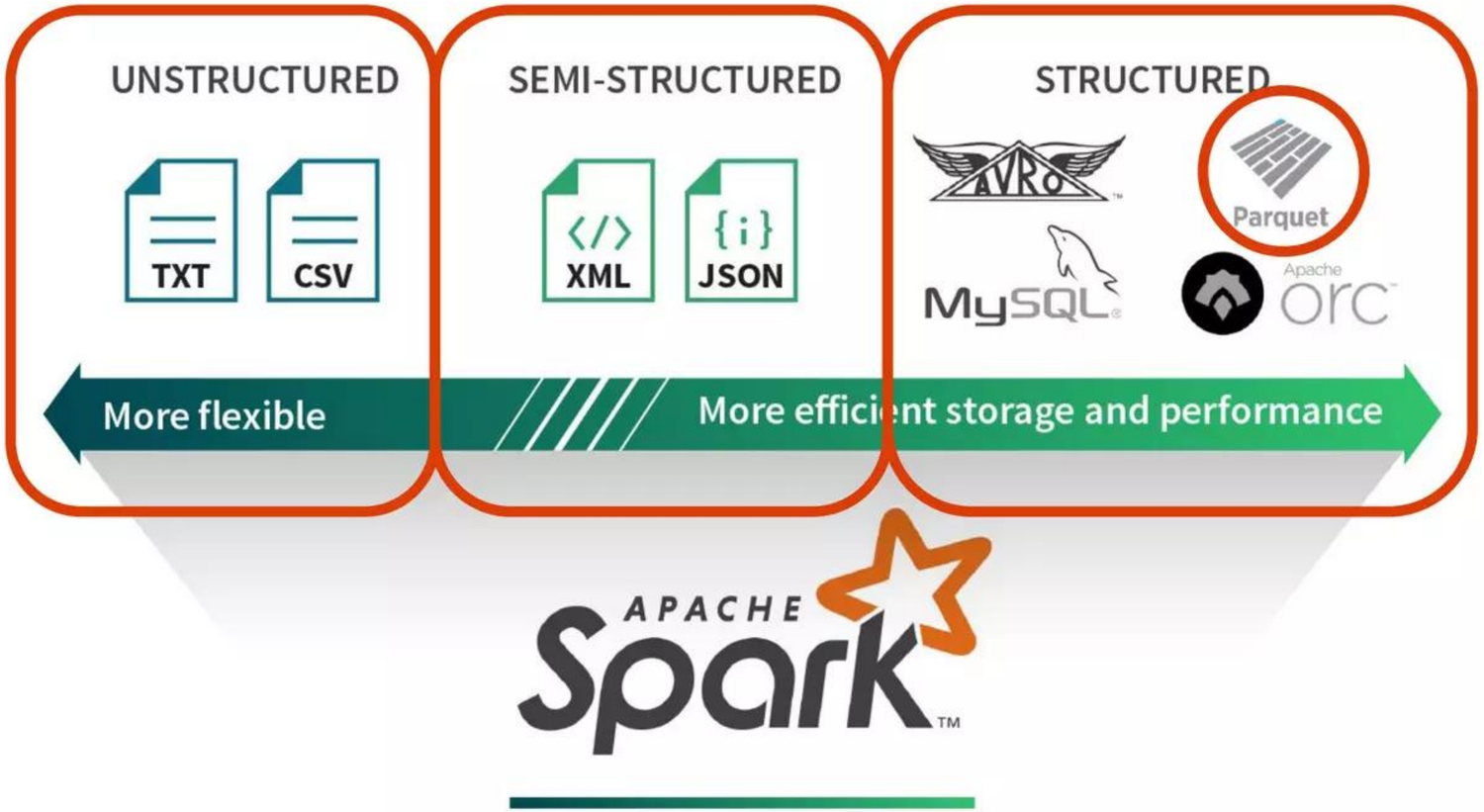
Data processing and analytics



Overview

- Data storage models
- The Parquet format
- Optimization opportunities

Data sources and formats

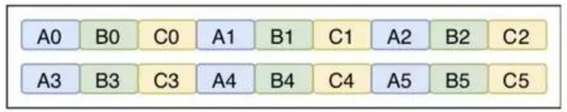


Physical storage layout models

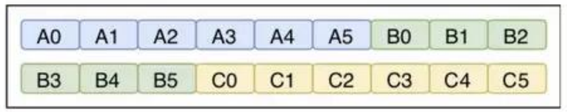
Logical

	Col A	Col B	Col C
Row 0	A0	B0	C0
Row 1	A1	B1	C1
Row 2	A2	B2	C2
Row 3	A3	B3	C3
Row 4	A4	B4	C4
Row 5	A5	B5	C5

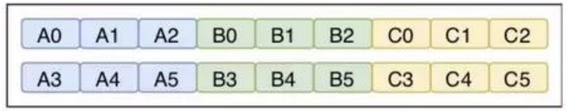
Physical



Row-wise



Columnar



Hybrid

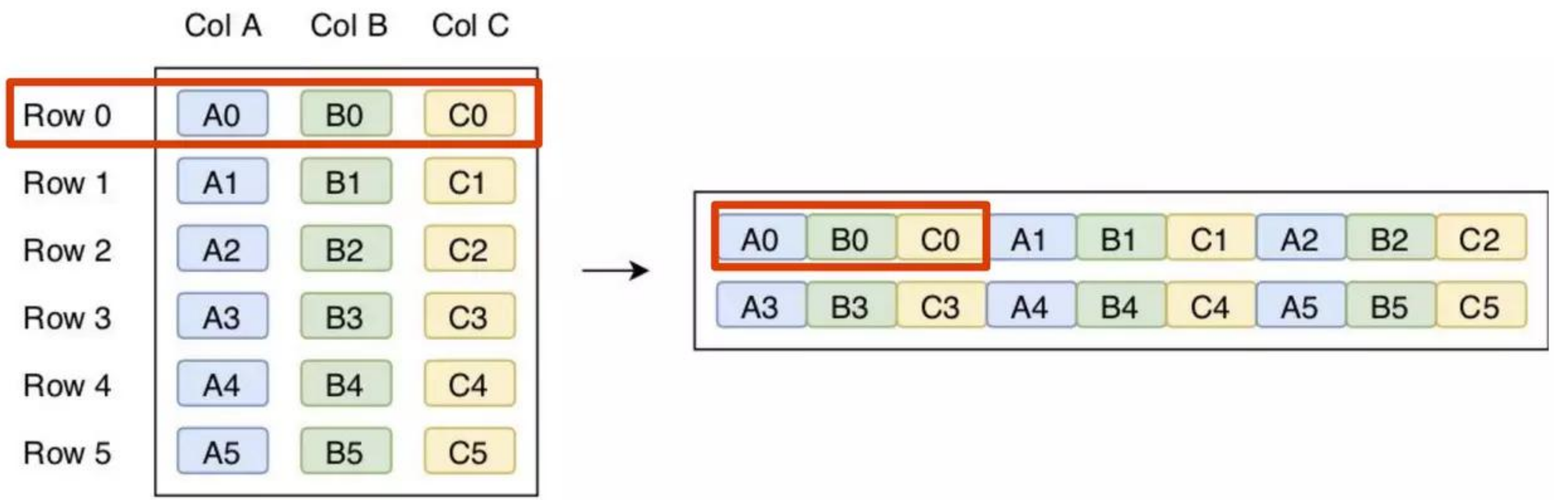
Different workloads

- OLTP
  - Online transaction processing
  - Lots of small operations involving *whole rows*

- OLAP
  - Online analytical processing
  - Few large operations involving *subset of all columns*

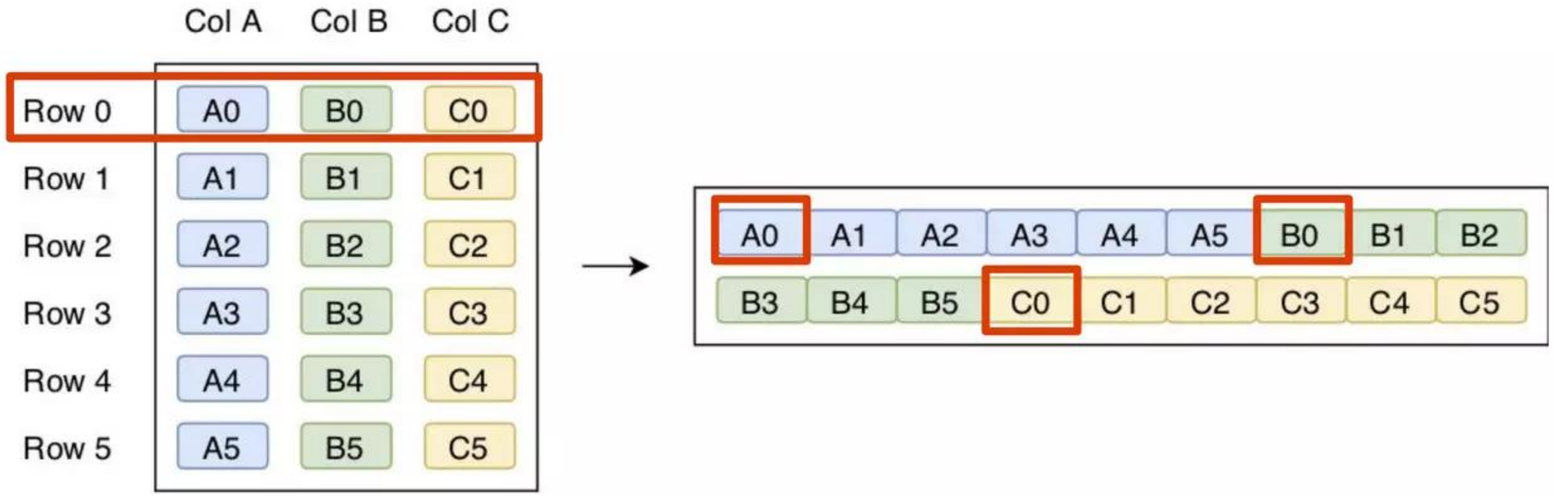
- Assumption: I/O is expensive (memory, disk, network..)

Row-wise



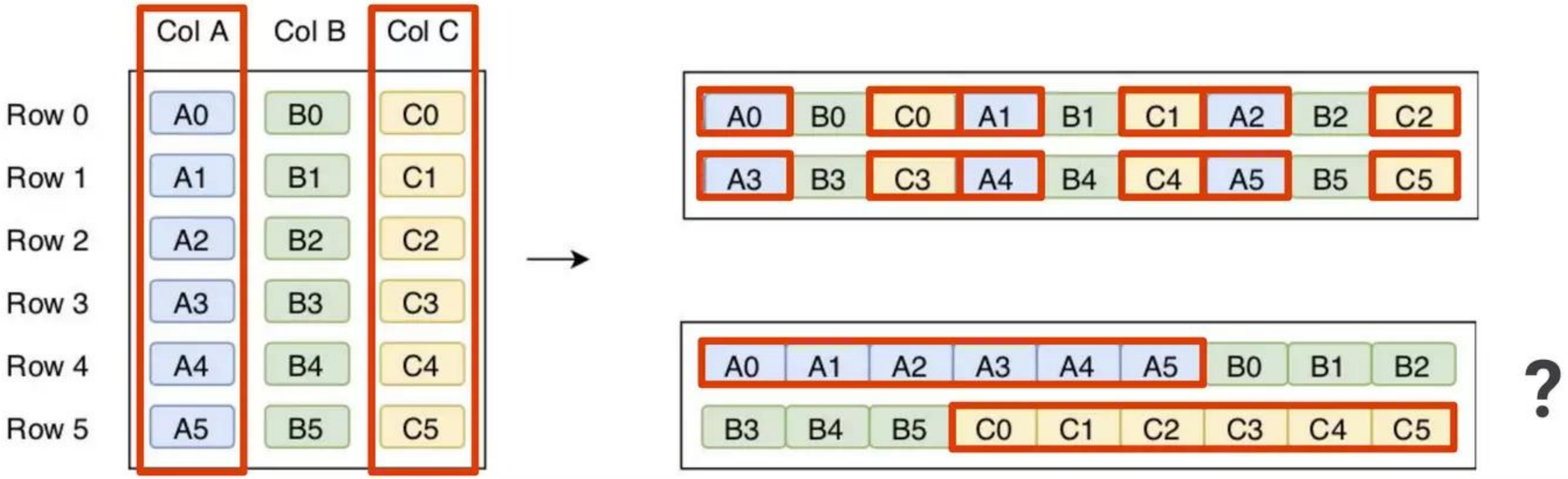
- Horizontal partitioning
- OLTP ✔, OLAP ❌

Columnar

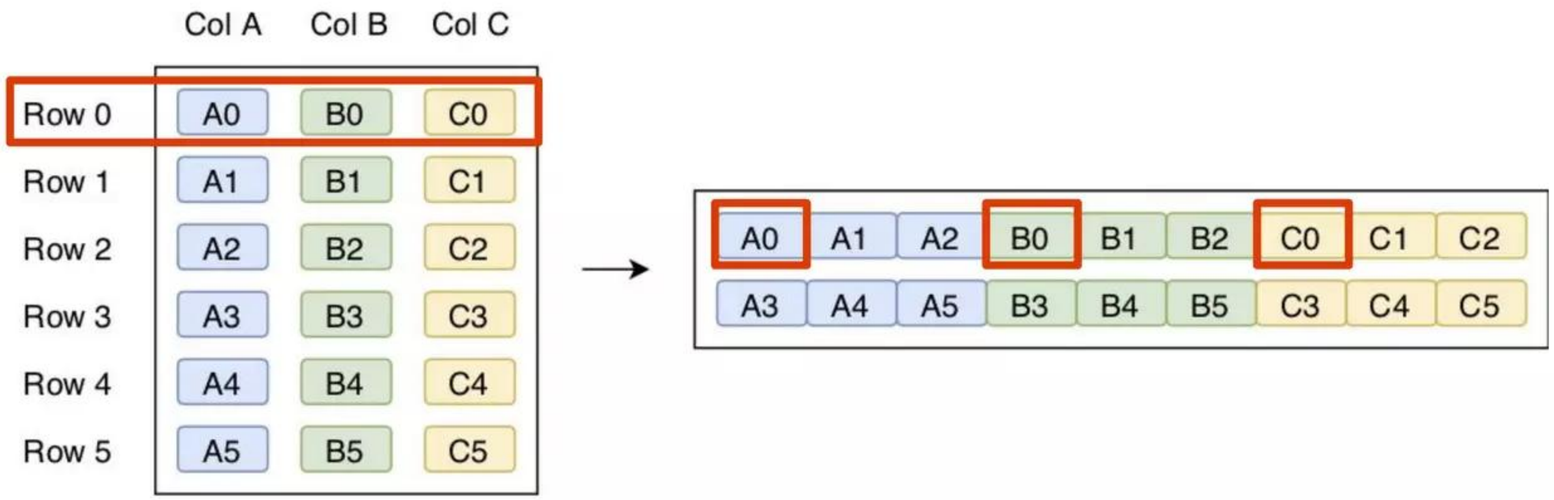


- Vertical partitioning
- OLTP ❌, OLAP ✅
  - Free projection pushdown
  - Compression opportunities

Row-wise vs Columnar



Hybrid



- Horizontal & vertical partitioning
- Used by Parquet & ORC
- Best of both worlds

Apache Parquet

- Initial effort by Twitter & Cloudera
- Open source storage format
  - Hybrid storage model (PAX)
- Widely used in Spark/Hadoop ecosystem
- One of the primary formats used by Databricks customers

Parquet: files

- On disk usually not a single file
- Logical file is defined by a root directory
  - Root dir contains one or multiple files

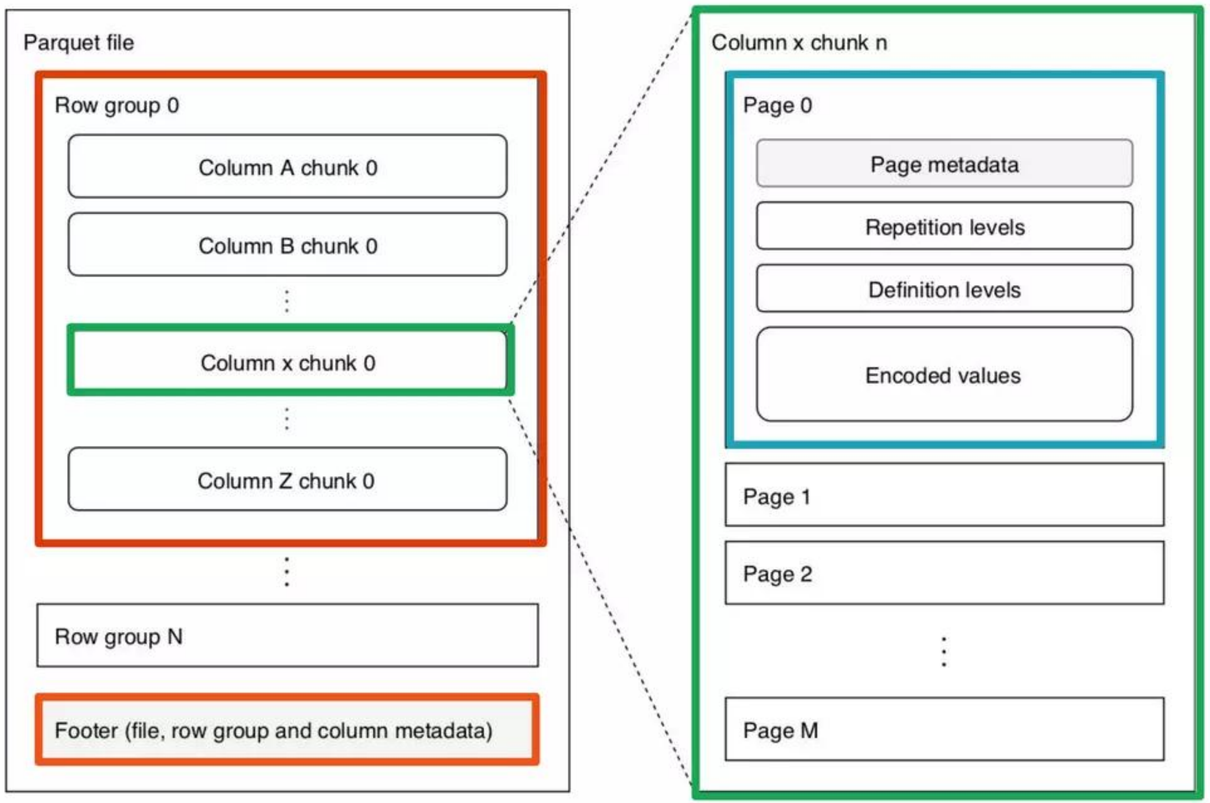
```
./example_parquet_file/
./example_parquet_file/part-00000-87439b68-7536-44a2-9eaa-1b40a236163d-c000.snappy.parquet
./example_parquet_file/part-00001-ae3c183b-d89d-4005-a3c0-c7df9a8e1f94-c000.snappy.parquet
```

  - **or** contains sub-directory structure with files in leaf directories

```
./example_parquet_file/
./example_parquet_file/country=Netherlands/
./example_parquet_file/country=Netherlands/part-00000-...-475b15e2874d.c000.snappy.parquet
./example_parquet_file/country=Netherlands/part-00001-...-c7df9a8e1f94.c000.snappy.parquet
```

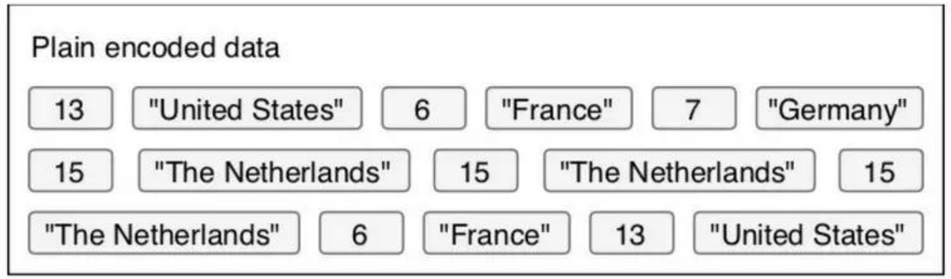
Parquet: data organization

- Data organization
  - Row-groups (*default 128MB*)
  - Column chunks
  - Pages (*default 1MB*)
    - Metadata
      - Min
      - Max
      - Count
    - Rep/def levels
    - Encoded values



Parquet: encoding schemes

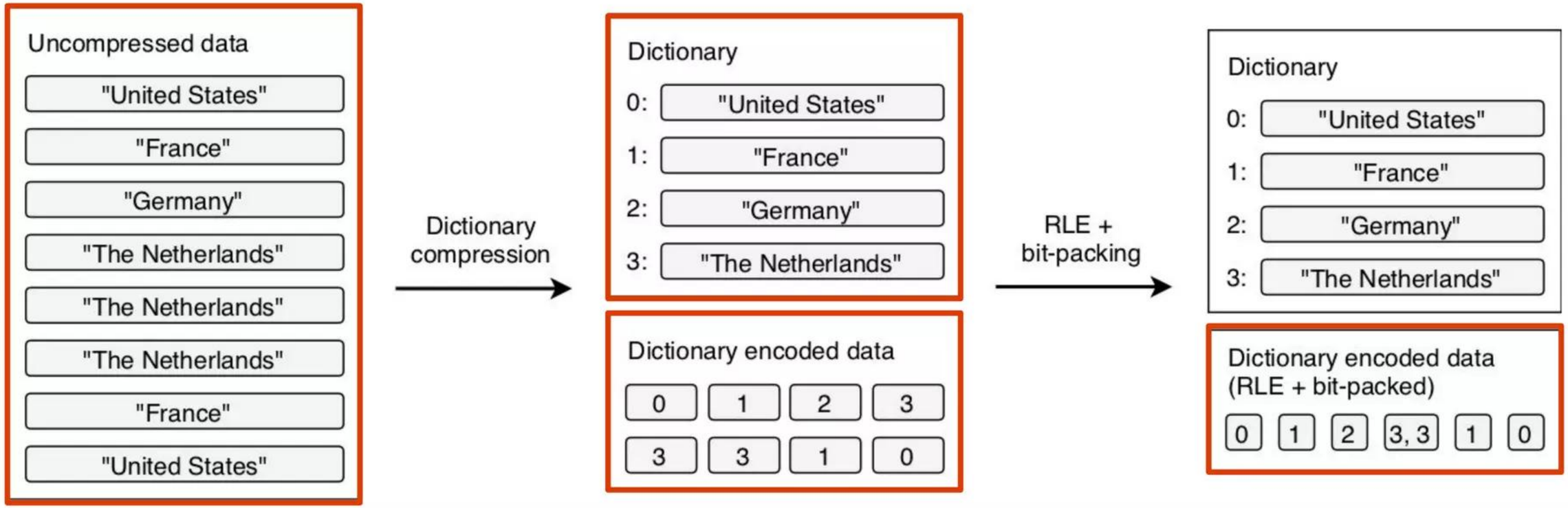
- **`PLAIN`**
  - Fixed-width: back-to-back
  - Non fixed-width: length prefixed



- **`RLE_DICTIONARY`**
  - Run-length encoding + bit-packing + dictionary compression
  - Assumes duplicate and repeated values

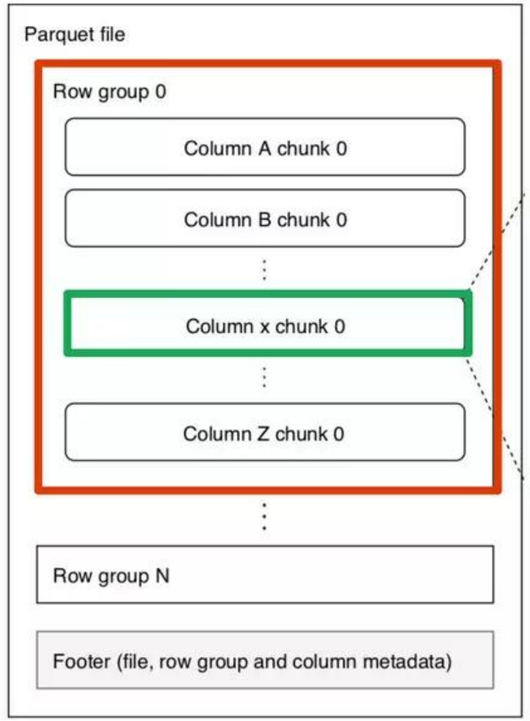
Parquet: encoding schemes

- `RLE_DICTIONARY`



Optimization: dictionary encoding

- Smaller files means less I/O
- Note: single dictionary per column chunk, size limit



Dictionary too big?

Automatic fallback to `PLAIN`...

Optimization: dictionary encoding

- Increase max dictionary size

  `parquet.dictionary.page.size`

- Decrease row-group size

  `parquet.block.size`

Optimization: dictionary encoding

- Inspect Parquet files using `parquet-tools`

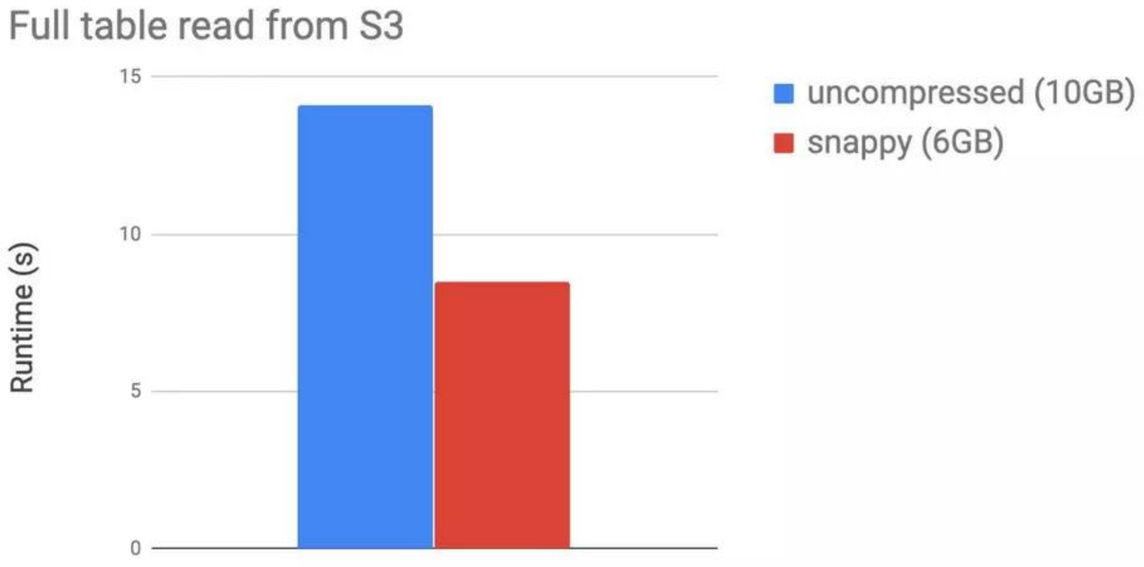
```
boudewijnbraams /Users/boudewijnbraams/Development/parquetdata
> parquet-tools dump -d -n 1largefile/part-00000-tid-3650625392277030654-d88abce6-09d9-46c1-8306-41149f65f6d9-8-1-c000.snappy.parquet
row group 0
--------------------------------------------------------------------------------
id:  INT64 SNAPPY DO:0 FPO:4 SZ:133783793/267378139/2.00 VC:33420100 ENC:PLAIN,BIT_PACKED ST:[min: 0, max: 33420099, num_nulls: 0]

    id TV=33420100 RL=0 DL=0
    ----------------------------------------------------------------------------
    page 0:    DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 0, max: 131100, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
    page 1:    DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 131101, max: 262201, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
    page 2:    DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 262202, max: 393302, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
    page 3:    DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 393303, max: 524403, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
    page 4:    DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 524404, max: 655504, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
    page 5:    DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 655505, max: 786605, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
    page 6:    DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 786606, max: 917706, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
    page 7:    DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 917707, max: 1048807, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
    page 8:    DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 1048808, max: 1179908, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
    page 9:    DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 1179909, max: 1311009, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
    page 10:   DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 1311010, max: 1442110, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
    page 11:   DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 1442111, max: 1573211, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
    page 12:   DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 1573212, max: 1704312, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
    page 13:   DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 1704313, max: 1835413, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
    page 14:   DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 1835414, max: 1966514, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
    page 15:   DLE:BIT_PACKED RLE:BIT_PACKED VLE:PLAIN ST:[min: 1966515, max: 2097615, num_nulls: 0] CRC:0 SZ:1048808 VC:131101
```

Optimization: page compression

- Compression of entire pages
  - Compression schemes (snappy, gzip, lzo…)

    `spark.sql.parquet.compression.codec`
  - Decompression speed vs I/O savings trade-off



Optimization: predicate pushdown

```
SELECT * FROM table WHERE x > 5
```

```
Row-group 0: x: [min: 0, max: 9]
Row-group 1: x: [min: 3, max: 7]
Row-group 2: x: [min: 1, max: 4]
…
```

- Leverage min/max statistics

  `spark.sql.parquet.filterPushdown`

Optimization: predicate pushdown

- Doesn’t work well on unsorted data
  - Large value range within row-group, low min, high max
  - What to do? Pre-sort data on predicate columns
- Use typed predicates
  - Match predicate and column type, don’t rely on casting/conversions
  - Example: use actual longs in predicate instead of ints for long columns

Optimization: predicate pushdown

```
SELECT * FROM table WHERE x = 5
```

```
Row-group 0: x: [min: 0, max: 9]
Row-group 1: x: [min: 3, max: 7]
Row-group 2: x: [min: 1, max: 4]
…
```

- Dictionary filtering!

  `parquet.filter.dictionary.enabled`

Optimization: partitioning

- Embed predicates in directory structure

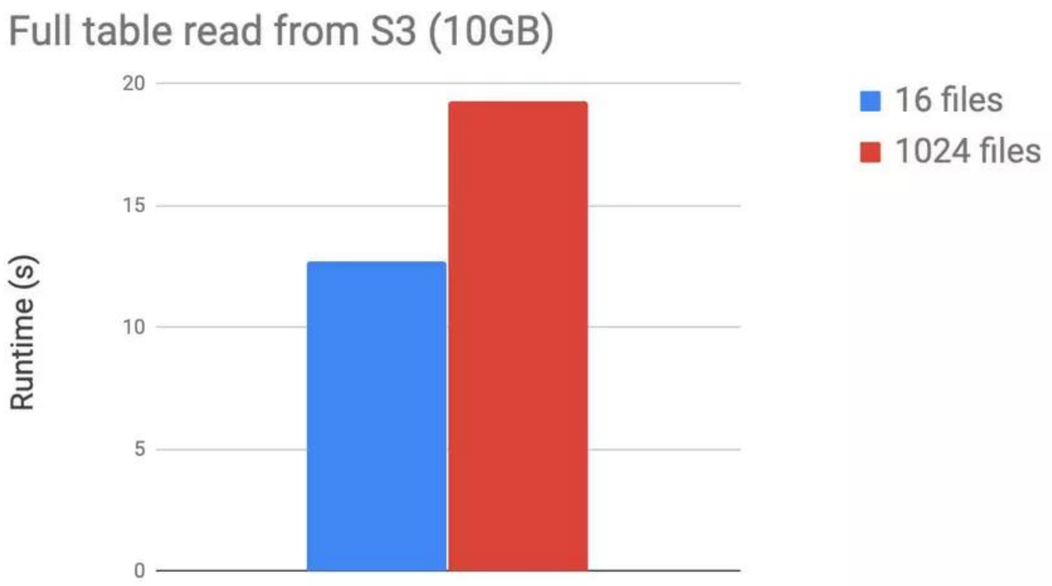
```
df.write.partitionBy("date").parquet(...)

./example_parquet_file/date=2019-10-15/...
./example_parquet_file/date=2019-10-16/...
./example_parquet_file/date=2019-10-17/part-00000-...-475b15e2874d.c000.snappy.parquet
```

…

Optimization: avoid many small files

- For every file
  - Set up internal data structures
  - Instantiate reader objects
  - Fetch file
  - Parse Parquet metadata



Optimization: avoid many small files

- Manual compaction

```
df.repartition(numPartitions).write.parquet(...)
```

or

```
df.coalesce(numPartitions).write.parquet(...)
```

- Watch out for incremental workload output!

Optimization: avoid few huge files

- Also avoid having huge files!
- `SELECT count(*)` on 250GB dataset
  - 250 partitions (~1GB each)
    - 5 mins
  - 1 huge partition (250GB)
    - **1 hour**
- Footer processing not optimized for speed...

Optimization: avoid many small files

- Manual repartitioning
  - Can we automate this optimization?
  - What about concurrent access?
- We need isolation of operations (i.e. ACID transactions)
- Is there anything for Spark and Parquet that we can use?

DELTA LAKE

Optimization: Delta Lake

- Open-source storage layer on top of Parquet in Spark
  - ACID transactions
  - Time travel (versioning via WAL)
  - ...

- Automated repartitioning (Databricks)
  - (Auto-) OPTIMIZE
  - Additional file-level skipping stats
    - Metadata stored in Parquet format, scalable
  - Z-ORDER clustering

DELTA LAKE

Conclusion

- Reduce I/O
  - Reduce size
    - Use page compression, accommodate for RLE_DICTIONARY
  - Avoid reading irrelevant data
    - Row-group skipping: min/max & dictionary filtering
    - Leverage Parquet partitioning
- Reduce overhead
  - Avoid having many small files (or a few huge)
- Delta Lake
  - (Auto-) OPTIMIZE, additional skipping, Z-ORDER

Thank you

databricks™

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