Big Data File Formats

VegasPy

11/14/2023

by Brett Kishkis

https://github.com/kishstats/big-data-file-formats

Why Big Data File Formats?

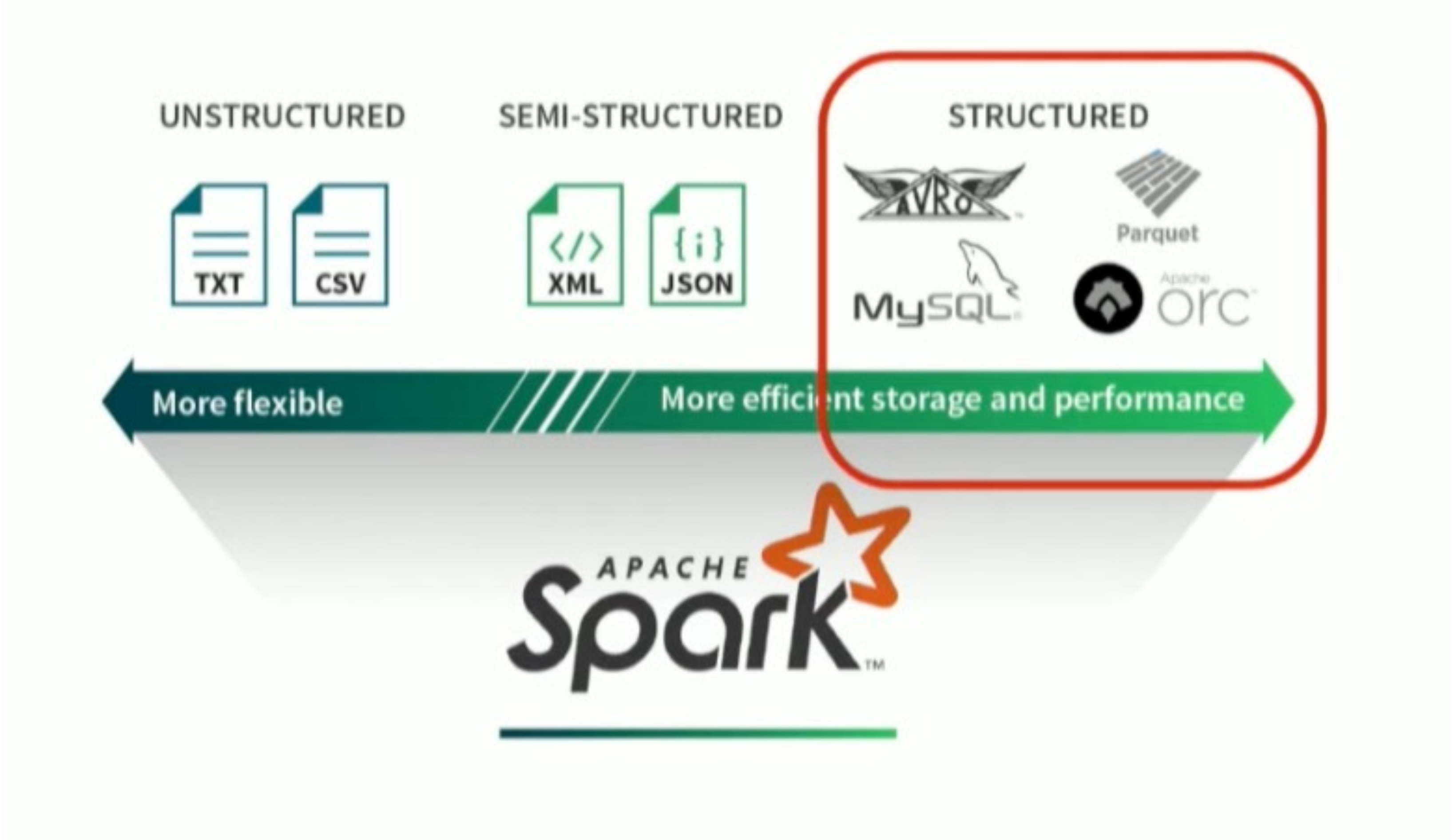
- Benefits
  - ↓Storage
  - ↓Costs
  - ↑Better Performance
    - Analyze faster
    - Train models faster

Parquet

Train Model

~~6 Hours 30 Minutes~~

Query performance is a function of how the data is stored.



[The Parquet Format and Performance Optimization Opportunities]()

CSV Files

- Benefits
  - Simplicity
  - Human readable
  - Mutable
  - Compact
    - Headers only written once
- Drawbacks
  - Does NOT support complex data structures
  - No universal method
  - No schema
    - Data types must be inferred

JSON Files

- Benefits
  - Hierarchical structures
  - Supports Arrays/Objects
  - Human readable
  - De-facto standard for REST API's
  - Similar structure to many NoSQL DB's
- Drawbacks
  - NOT compact
    - Repeated keys --> more memory
  - No Support for Streaming/Splitting
    - Fully received, then parsed
  - No Schema enforcement

{ JSON }

Parquet

- Columnar Storage
  - Column Subsets (Column Pruning)
- Compression
- Metadata
  - Schema
    - File Footer
- Use cases
  - WORM (Write Once Read Many)
- Predicate Pushdown
  - Filtering
- Platforms
  - Spark
- *Does NOT natively support ACID transactions

```
df = pd.read_parquet(
  '/file',
  columns=['ticker', 'company_name', 'closing_price'])
```

Column Subsets Example

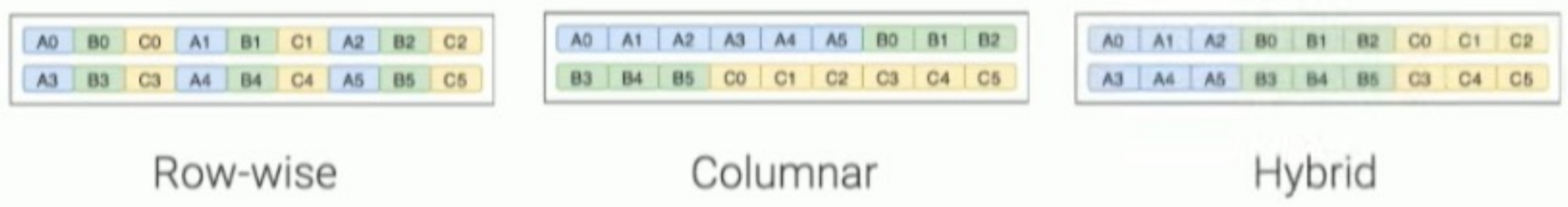
**Delta Lake, an open-source storage layer supports ACID transactions and uses Parquet as its file format.*

DELTA LAKE

Row vs Columnar Format

- Columnar Format
  - Storing data column by column
  - Read optimized
- Row Format
  - Storing data row by row
  - Write optimized for transactions
    - Streaming
  - Similar to traditional DB

- OLAP
  - Online Analytical Processing
  - Queries for subsets of columns
- OLTP
  - Online Transaction Processing
  - Executing transactions for entire rows



OLTP✓ OLAP✓

Parquet

- Predicate Pushdown filtering
    - Row subsets
    - Similar to WHERE clause
    - Reduces the amount of data read from the file
    - Recommended:
        - Sort data before writing to file

```
df = pd.read_parquet(
  'path/to/file',
  filters=[[('sector', '=', 'Information Technology')]])
```

Parquet Example Case

Use case: find the average market cap of all NASDAQ stocks from the `stocks` table.

Ticker	Name	Price	Change	Change Pct	Market Cap	P/E	Exchange
AAPL	Apple Inc	$186.40	4.23	2.32%	$2,899 B	30.39	NASDAQ
OLO	Olo Inc	$4.72	0.22	5.01%	$1 B	-	NYSE
LULU	Lululemon Athletica Inc	$413.67	7.09	1.74%	$52 B	52.39	NASDAQ
BLZE	Backblaze Inc	$5.75	0.18	3.23%	$215 M	-	NASDAQ
BROS	Dutch Bros Inc	$27.53	0.43	1.59%	$5 B	709.72	NYSE

Column Pruning:
- system only needs to read the "exchange" and "market_cap" columns
  - not the entire dataset

Predicate Pushdown:
- (exchange = 'NASDAQ')
- skip over all the rows where the "exchange" is not "NASDAQ"

Parquet

- Splitting into Multiple Files
- Root directory
    - Contains a collection of files
    - Can also contain subdirectories
- Partitioning
    - Parquet (Part) Files
    - Metadata Files
    - _metadata
        - quickly understand the structure and statistics of the data without scanning all the files (Spark)
    - _SUCCESS
        - successful completion of the write operation
    - Partition Directories
        - directory structure that reflects the partitioning scheme
        - i.e. partition by a column named `date`

```
/path/to/output/folder/
    part-00000-uuid.c000.snappy.parquet
    part-00001-uuid.c000.snappy.parquet
    ...
    _SUCCESS
    _common_metadata (optional)
    _metadata
    .part-00000-uuid.c000.snappy.parquet.crc (optional)
    ...
    /date=2021-01-01/
        part-00000-uuid.c000.snappy.parquet
        ...
    /date=2021-01-02/
        part-00000-uuid.c000.snappy.parquet
        ...
```

Parquet Partitioning

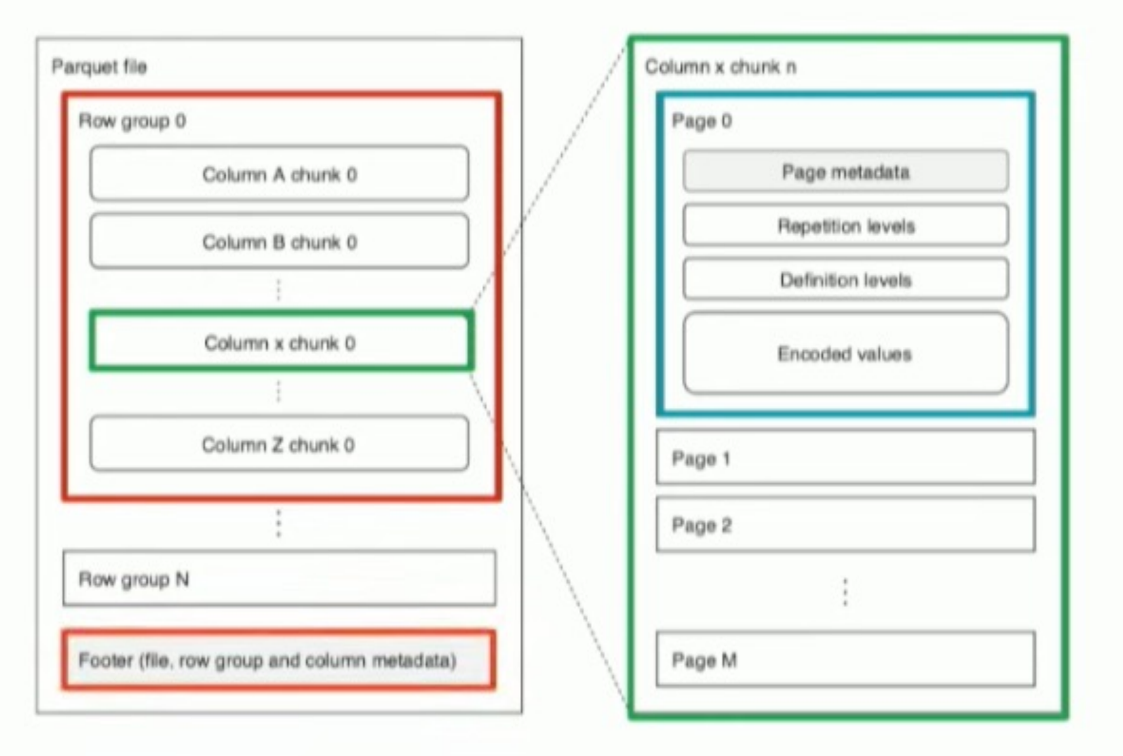
```
df.write.partitionBy("date").parquet(...)

./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
```

Spark dataframe partitioning by date column

- useful if you know ahead of time what your predicates will be

Parquet File Structure

- Row-groups
- Column chunks
- Pages
  - Metadata
    - Min
    - Max
    - Count
  - Rep/def levels
  - Encoded values

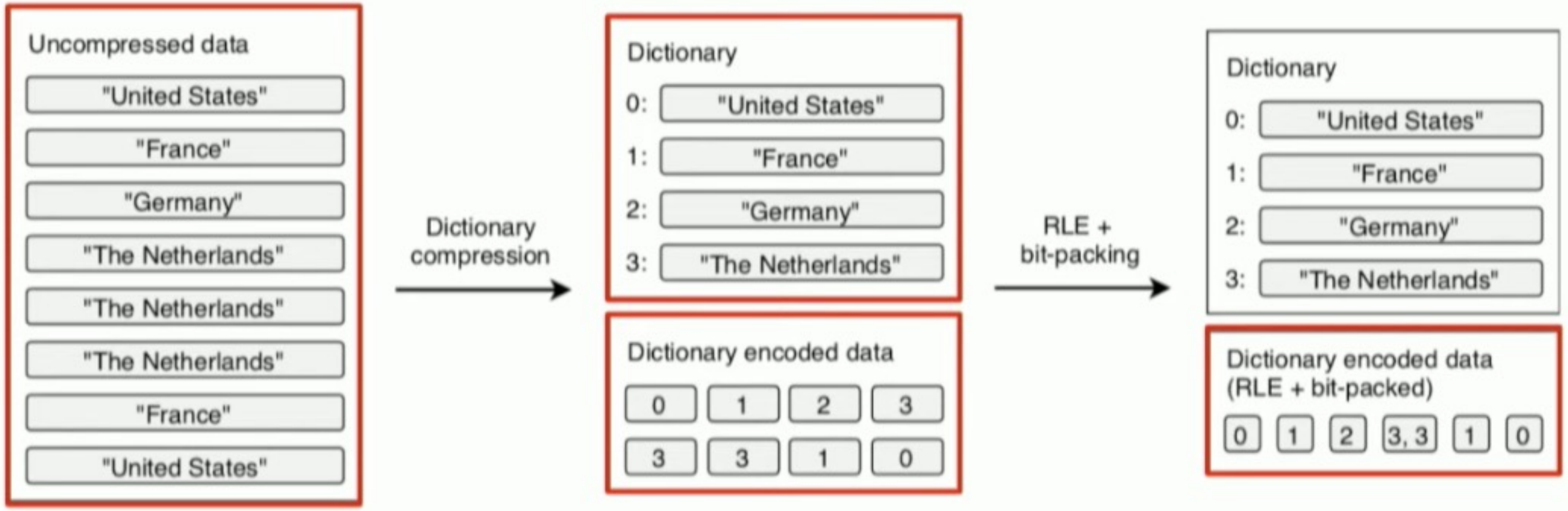


See "The Parquet Format and Performance Optimization Opportunities" video

Parquet File Encoding

- PLAIN
- RLE_DICTIONARY
- Others
  - https://parquet.apache.org/docs/file-format/data-pages/encodings/

RLE Dictionary Encoding



See "The Parquet Format and Performance Optimization Opportunities" video

Parquet File Compression

- Things to consider:
  - Trade-offs between storage space, I/O bandwidth, and CPU usage
- Compression Formats
  - Snappy
    - Most common
    - Generally the default
    - Good balance between compression ratio and speed
  - GZip
    - better compression ratios at the expense of speed and CPU
  - LZO
  - Others
  - Can have ‘None’
- Some codecs may require additional dependencies to be installed
- **Parquet compression definitions**

Using Parquet with Python

- Pandas
  - data structures and data analysis tools
  - `read_parquet` method
  - `to_parquet` dataframe method
- pyarrow
  - Read metadata
  - Properties to inspect schema and metadata
  - Create Parquet files
- pyspark
  - Python for Apache Spark
  - Big data and analytics processing

“A cross-language development platform for in-memory analytics”

Apache ORC

- Optimized Row Columnar (ORC)
- Steaming Writes
- ACID Transactions
- Platforms
  - Apache Hive
  - Hadoop

Apache ORC vs Parquet

- Many Similarities
- Apache ORC better for:
  - Write-heavy workloads
  - ACID transactions
- Parquet better for:
  - Read-heavy workloads
  - Queries for aggregate data
  - Community support
- Platform Choice
  - Spark vs Hive

Using Apache ORC with Python

- Pandas
  - data structures and data analysis tools
  - `read_orc` method
  - `to_orc` dataframe method
- pyarrow
  - Read metadata
  - Properties to inspect schema and metadata
  - Create ORC files

"A cross-language development platform for in-memory analytics"

- Row-oriented
- Schema based
  - dynamic schema evolution
  - written in JSON
- Serialization
  - JSON (metadata)
  - Binary (data)
- Splitting
  - Data into subsets
  - Parallel Processing
- Steaming
- Platforms
  - Kafka

Using Avro with Python

- fastavro
  - Better performance for CPython
- avro
  - Officially supported module
- Pandas (with pandavro)
  - pandavro is an interface between AVRO and Pandas
  - `read_avro` pandavro method
  - `to_avro` pandavro method

fastavro Example

```python
from fastavro import reader, writer, parse_schema

# Reading an Avro file
with open('path/to/file.avro', 'rb') as f:
    avro_reader = reader(f)
    for record in avro_reader:
        print(record)

# Writing to an Avro file
schema = {...}  # Define your Avro schema here
parsed_schema = parse_schema(schema)

records = [...]  # Your data records
with open('path/to/output.avro', 'wb') as f:
    writer(f, parsed_schema, records)
```

References

- Parquet
  - [The Parquet Format and Performance Optimization Opportunities Boudewijn Braams (Databricks)]
  - [Netflix Dataset Performance Test]
  - [Parquet Performance Tests (using Netflix Dataset)]
  - [Why Parquet vs. ORC: An In-depth Comparison of File Formats]