Files Formats – not just CSV

- Key factor in Big Data processing and query performance
- Schema Evolution
- Compression and Splittability
- Data Processing
  - Write performance
  - Partial read
  - Full read

Optimizations – CPU and I/O

Statistics for filtering and query optimization

projection push down

X	Y	Z
x1	y1	z1
x2	y2	z2
x3	y3	z3
x4	y4	z4
x5	y5	z5

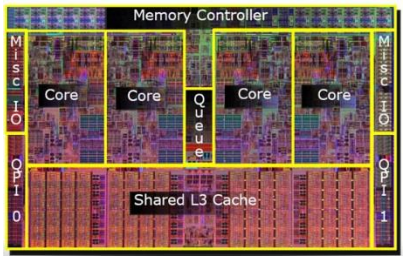
predicate push down

X	Y	Z
x1	y1	z1
x2	y2	z2
x3	y3	z3
x4	y4	z4
x5	y5	z5

read only the data you need

X	Y	Z
x1	y1	z1
x2	y2	z2
x3	y3	z3
x4	y4	z4
x5	y5	z5

Minimizes CPU cache misses



SLOW

cache misses costs cpu cycles

Parquet

- columnar storage format
- key strength is to store nested data in truly columnar format using definition and repetition levels[1]

Nested schema

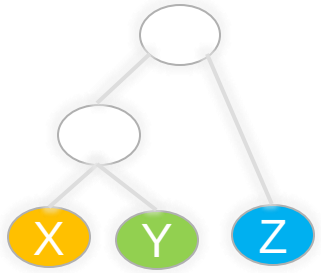
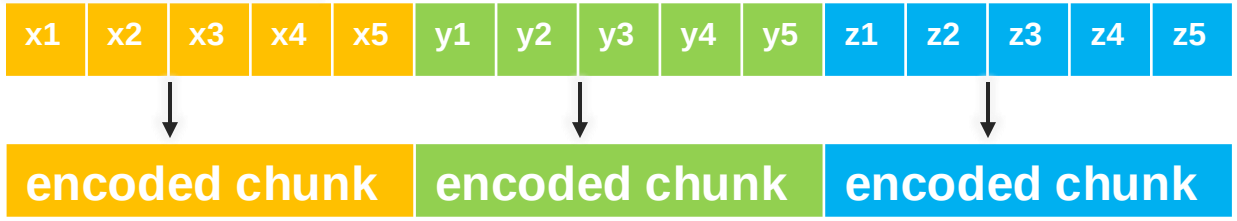


Table representation

X	Y	Z
x1	y1	z1
x2	y2	z2
x3	y3	z3
x4	y4	z4
x5	Y5	z5

Row format

Columnar format



(1) Dremel made simple with parquet - https://blog.twitter.com/2013/dremel-made-simple-with-parquet

What is a Parquet File?

Apache Parquet is a columnar storage file format designed for efficient data processing and analytics.

- **Key Features**:
  - **Columnar Format**: Stores data by columns, not rows.
  - **Optimized for Big Data**: Ideal for distributed systems like Apache Hadoop, Spark, and Hive.
  - **Compression**: Supports efficient compression and encoding schemes to reduce file size.

- **Advantages**:
  1. **Efficient I/O**: Faster queries by reading only required columns.
  2. **Small File Size**: Highly compressed, saving storage.
  3. **Schema Evolution**: Supports changes to schema over time.
  4. **Interoperability**: Works seamlessly with data tools like Hive, Spark, and Pandas.

Key Features of Parquet

- **Columnar Storage**:
  - Ideal for analytical workloads where specific columns are queried frequently.
- **Data Compression**:
  - Reduces storage using advanced encoding like Snappy, GZIP, and LZO.
- **Metadata Support**:
  - Stores schema and statistics for efficient querying.
- **Language-Independent**:
  - Can be used with Java, Python, C++, and more.

- **Parquet vs Other Formats**

Feature	Parquet	CSV	JSON	Avro
File Size	Small	Large	Medium	Small
Read Efficiency	High	Low	Medium	Medium
Schema Support	Yes	No	No	Yes
Compression	Advanced	None	None	Basic

Common Use Cases

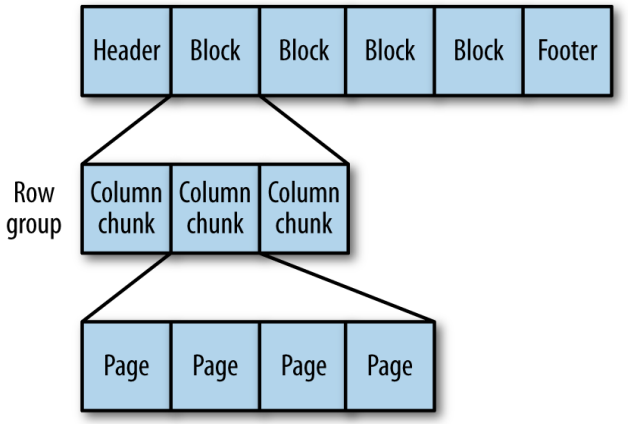
1. **Big Data Analytics**:
   - Storing and processing large datasets with tools like Hadoop and Spark.
2. **Data Warehousing**:
   - Optimized for OLAP workloads.
3. **Cloud Storage**:
   - Supported by AWS S3, Google Cloud Storage, and Azure.
4. **ETL Pipelines**:
   - Efficient for extracting, transforming, and loading data.

- **Read/Write Parquet Files**:
  - **Python (Pandas)**: import pandas as pd df = pd.read_parquet('file.parquet') # Reading df.to_parquet('output.parquet') # Writing
  - **Apache Spark**: spark.read.parquet("file.parquet") # Reading df.write.parquet("output.parquet") # Writing

- **Storage Tools**:
  - Hive, Presto, and Snowflake support native Parquet formats.

Parquet file structure & Configuration

Internal structure of parquet file



Configurable parquet parameters

Property name	Default value	Description
parquet.block.size	128 MB	The size in bytes of a block (row group).
parquet.page.size	1MB	The size in bytes of a page.
parquet.dictionary.page.size	1MB	The maximum allowed size in bytes of a dictionary before falling back to plain encoding for a page.
parquet.enable.dictionary	true	Whether to use dictionary encoding.
parquet.compression	UNCOMPRESSED	The type of compression: UNCOMPRESSED, SNAPPY, GZIP & LZO

Parquet is state-of-the-art, open-source columnar format the supports *most* of Hadoop processing frameworks and is optimized for high compression and high scan efficiency

Encoding

- Delta Encoding:
  - E.g timestamp can be encoded by storing first value and the delta between subsequent values which tend to be small due to temporal validity
- Prefix Encoding:
  - delta encoding for strings
- Dictionary Encoding:
  - Small set of values, e.g post code, ip addresses etc
- Run Length Encoding:
  - repeating data

Delta Encoding:

- **Definition**: Delta encoding reduces storage requirements by encoding data as the difference (or "delta") between sequential values instead of storing the actual values.
- **Example**:
  - Given timestamps: [1000, 1005, 1010, 1015]
  - Instead of storing all timestamps, delta encoding stores:
    - First value: 1000
    - Differences: [+5, +5, +5]
  - This reduces redundancy because differences are typically smaller in magnitude, allowing for more efficient compression.
- **Use Case**: Commonly used in time-series data, where values are sequential and changes between consecutive data points are small.

Prefix Encoding:

- **Definition**: A variation of delta encoding, applied specifically to strings. It identifies common prefixes and replaces them with a shorter representation.
- **Example**:
  - Strings: ["apple", "applet", "applesauce"]
  - Encoding:
    - Base: "apple"
    - Additions: ["", "t", "sauce"]
  - This encodes only the differences from the base string.
- **Use Case**: Effective in applications with similar string data, such as filenames, URLs, or text with repetitive phrases.

Dictionary Encoding:

- **Definition**: Compresses data by replacing repeated values with a reference to an entry in a dictionary (a predefined mapping of values to unique identifiers).
- **Example**:
  - Data: ["CA", "NY", "CA", "CA", "TX"]
  - Dictionary:
    - "CA" -> 1
    - "NY" -> 2
    - "TX" -> 3
  - Encoded Data: [1, 2, 1, 1, 3]
- **Use Case**: Useful for categorical data with a limited set of unique values, such as postal codes, country codes, or IP addresses.

Run-Length Encoding (RLE):

- **Definition**: Compresses data by replacing consecutive repeated values with a single value and a count of repetitions.

- **Example**:
  - Data: [A, A, A, B, B, C, C, C, C]
  - Encoded: [(A, 3), (B, 2), (C, 4)]

- **Use Case**: Effective for datasets with long runs of the same value, such as image data with large regions of the same color, or simple binary data with repeating patterns.

Comparison and Applications:

- **Delta Encoding** is ideal for ordered numeric data with small incremental changes, like sensor readings.
- **Prefix Encoding** is best for string data with common prefixes.
- **Dictionary Encoding** works well for datasets with repetitive discrete values.
- **Run-Length Encoding** excels with repetitive sequences in data, especially in multimedia compression.

AVRO

- Language neutral data serialization system
  - Write a file in python and read it in C
- AVRO data is described using language independent schema
- AVRO schemas are usually written in JSON and data is encoded in binary format
- Supports schema evolution
  - producers and consumers at different versions of schema
- Supports compression and are splittable

Avro – File structure and example

Sample AVRO schema in JSON format

```
{
  "type" : "record",
  "name" : "tweets",
  "fields" : [ {
    "name" : "username",
    "type" : "string",
}, {
    "name" : "tweet",
    "type" : "string",
}, {
    "name" : "timestamp",
    "type" : "long",
} ],
  "doc:" : "schema for storing tweets"
}
```

Avro file structure