01 Parquet

Apache Parquet

Introduction to Apache Parquet

What is Apache Parquet?

Apache Parquet is an **open-source, column-oriented data storage format** designed specifically for efficient data storage and retrieval in big data environments. Originally developed by Twitter and Cloudera, Parquet has become the de facto standard for analytical workloads in the big data ecosystem.

Key Design Principles

1. **Language Agnostic**: Supported across multiple programming languages (Java, Python, C++, etc.)
2. **Platform Independent**: Works with various big data frameworks (Spark, Hive, Impala, Drill)
3. **Self-Describing**: Contains schema information within the file
4. **Optimized for Analytics**: Designed for OLAP (Online Analytical Processing) workloads
5. **Efficient Compression**: Advanced encoding and compression techniques

Why Parquet Matters in Big Data

- **Storage Cost Reduction**: Significantly smaller file sizes compared to row-based formats
- **Query Performance**: Faster analytical queries through columnar access patterns
- **I/O Optimization**: Reduced network and disk I/O through efficient data layout
- **Ecosystem Integration**: Native support in most big data tools and cloud platforms

---

Columnar Storage Architecture

Row-Based vs. Column-Based Storage

Traditional Row-Based Storage

```
Record 1: [ID=1, Name="Alice", Age=25, Salary=50000]
Record 2: [ID=2, Name="Bob", Age=30, Salary=60000]
Record 3: [ID=3, Name="Charlie", Age=35, Salary=70000]
```

Parquet's Column-Based Storage

```
ID Column: [1, 2, 3]
Name Column: ["Alice", "Bob", "Charlie"]
Age Column: [25, 30, 35]
Salary Column: [50000, 60000, 70000]
```

Advantages of Columnar Storage

1. **Better Compression**: Similar data types stored together compress more efficiently
2. **Column Pruning**: Read only required columns, reducing I/O
3. **Vectorized Processing**: Modern CPUs can process columnar data more efficiently
4. **Cache Efficiency**: Better CPU cache utilization due to data locality
5. **Analytics Optimization**: Most analytical queries access subset of columns

Memory Layout Benefits

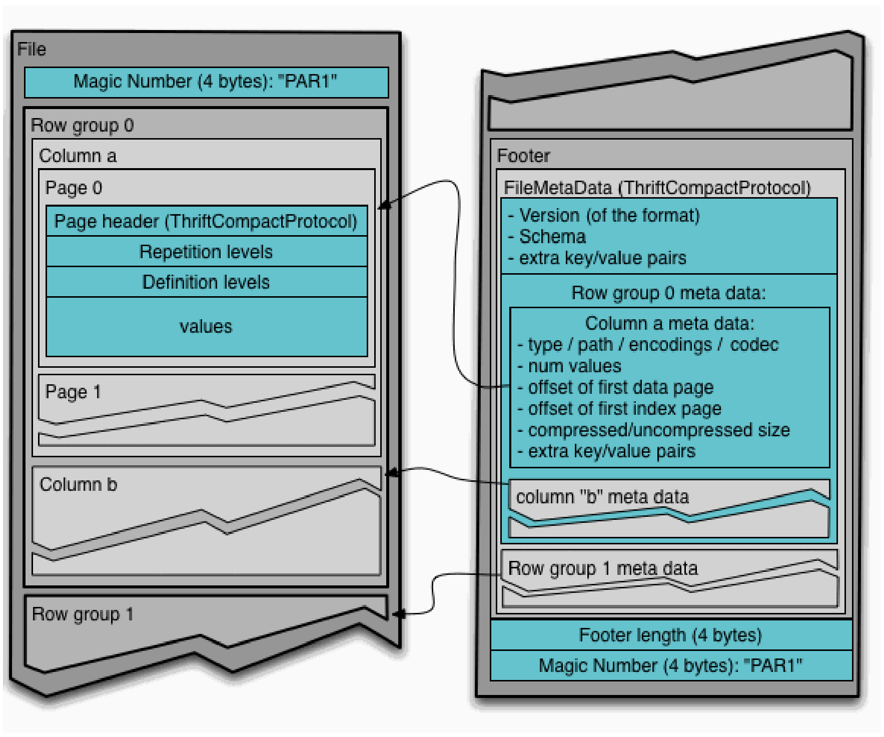
- **Spatial Locality**: Related data stored contiguously in memory
- **Temporal Locality**: Frequently accessed data remains in cache longer
- **SIMD Operations**: Single Instruction, Multiple Data operations on columns
- **Prefetching**: CPU can predict and prefetch columnar data patterns

---

File Structure and Organization

Hierarchical File Structure

```
Parquet File
├── File Header
│   ├── Magic Number (4 bytes: "PAR1")
│   └── File Metadata Pointer
├── Row Groups (1 to N)
│   ├── Row Group 1
│   │   ├── Column Chunk 1
│   │   │   ├── Data Pages
│   │   │   ├── Dictionary Page (optional)
│   │   │   └── Column Metadata
│   │   ├── Column Chunk 2
│   │   └── ...
│   └── Row Group N
└── File Footer
    ├── Row Group Metadata
    ├── Schema Definition
    ├── Key-Value Metadata
    └── Magic Number (4 bytes: "PAR1")
```


File
Magic Number (4 bytes): "PAR1"
Row group 0
Column a
Page 0
Page header (ThriftCompactProtocol)
Repetition levels
Definition levels
values
Page 1
Column b
Row group 1
Footer
FileMetaData (ThriftCompactProtocol)
- Version (of the format)
- Schema
- extra key/value pairs
Row group 0 meta data:
Column a meta data:
- type / path / encodings / codec
- num values
- offset of first data page
- offset of first index page
- compressed/uncompressed size
- extra key/value pairs
column "b" meta data
Row group 1 meta data
Footer length (4 bytes)
Magic Number (4 bytes): "PAR1"

File Format | Parquet

Row Groups

Definition: Horizontal partitions of data, typically 64MB to 1GB in size.

Characteristics:

- **Independent Processing**: Each row group can be processed separately
- **Parallel Access**: Multiple row groups can be read concurrently
- **Memory Management**: Size chosen to fit comfortably in memory
- **Fault Tolerance**: Corruption in one row group doesn't affect others

Column Chunks

Definition: All data for a single column within a row group.

Components:

- **Data Pages**: Actual column data with encoding/compression
- **Dictionary Page**: Optional page containing dictionary for encoding
- **Index Pages**: Optional pages for improved query performance
- **Metadata**: Statistics and encoding information

Data Pages

Structure:

- **Header**: Encoding type, compression, statistics
- **Repetition Levels**: Handle nested/repeated structures
- **Definition Levels**: Handle optional/null values
- **Values**: Actual encoded data

Page Types:

1. **Data Page V1**: Original page format
2. **Data Page V2**: Improved format with better compression
3. **Dictionary Page**: Contains dictionary values

4. **Index Page**: Contains page-level indices

Encoding and Compression Techniques

Encoding Schemes

1. Dictionary Encoding

Use Case: Columns with repeated values (categorical data)

Mechanism:

- Create dictionary of unique values
- Replace actual values with dictionary indices
- Store dictionary separately in the file

Example:

```
Original Data: ["Red", "Blue", "Red", "Green", "Blue", "Red"]
Dictionary: {0: "Red", 1: "Blue", 2: "Green"}
Encoded Data: [0, 1, 0, 2, 1, 0]
```

Benefits:

- Significant space savings for low-cardinality columns
- Faster string comparisons using integer indices
- Improved compression of dictionary indices

2. Run-Length Encoding (RLE)

Use Case: Columns with consecutive repeated values

Mechanism:

- Store value and count of consecutive occurrences
- Particularly effective for sorted data

Example:

```
Original Data: [1, 1, 1, 2, 2, 3, 3, 3, 3]
RLE Encoded: [(1, 3), (2, 2), (3, 4)]
```

3. Bit-Packing

Use Case: Integer columns with limited value ranges

Mechanism:

- Use minimum number of bits required to represent values
- Pack multiple values into single bytes

Example:

```
Values 0-7 require only 3 bits each
Original: 8 bytes for 8 integers
Bit-packed: 3 bytes for 8 integers
```

4. Delta Encoding

Use Case: Columns with incremental or trending values

Mechanism:

- Store first value and differences between consecutive values
- Effective for timestamps, IDs, sorted numeric data

Example:

```
Original: [100, 102, 105, 107, 110]
Delta: [100, 2, 3, 2, 3]
```

5. Plain Encoding

Use Case: When other encodings are not beneficial

Mechanism:

- Store values in their original binary representation
- No transformation applied

Compression Algorithms

1. Snappy

- **Speed**: Very fast compression/decompression
- **Ratio**: Moderate compression ratio
- **Use Case**: Real-time processing, interactive queries
- **CPU Usage**: Low CPU overhead

2. GZIP

- **Speed**: Slower than Snappy
- **Ratio**: Better compression ratio than Snappy
- **Use Case**: Storage optimization, archival data
- **CPU Usage**: Higher CPU overhead

3. LZO (Lempel-Ziv-Oberhumer)

- **Speed**: Fast compression, very fast decompression
- **Ratio**: Similar to Snappy
- **Use Case**: Balanced performance and compression
- **Splittable**: Supports splitting for parallel processing

4. ZSTD (Zstandard)

- **Speed**: Good balance of speed and compression
- **Ratio**: Better than Snappy, competitive with GZIP
- **Use Case**: Modern applications requiring good performance
- **Adaptive**: Automatic parameter tuning

5. Brotli

- **Speed**: Slower compression, fast decompression
- **Ratio**: Excellent compression ratio
- **Use Case**: Long-term storage, bandwidth-limited environments

Encoding Selection Strategy

Automatic Selection: Parquet writers analyze data characteristics and automatically select optimal encoding:

1. **Data Type Analysis**: Different encodings for different types
2. **Cardinality Assessment**: Dictionary encoding for low cardinality
3. **Pattern Recognition**: RLE for repeated patterns
4. **Range Analysis**: Bit-packing for limited ranges
5. **Statistics Collection**: Guide encoding decisions

Size Optimization and Storage Efficiency

Why Parquet Files are Smaller

1. Columnar Storage Benefits

- **Type Homogeneity**: Similar data types compress better
- **Value Locality**: Similar values stored together
- **Null Compression**: Efficient handling of missing values
- **Pattern Recognition**: Easier to identify compression opportunities

2. Advanced Encoding Techniques

- **Dictionary Compression**: 50-90% reduction for categorical data
- **RLE Compression**: 90%+ reduction for repeated values
- **Bit-Packing**: 50-87% reduction for small integer ranges
- **Delta Encoding**: Significant reduction for sequential data

3. Schema Optimization

- **Nested Structure Handling**: Efficient storage of complex types
- **Optional Field Handling**: No storage overhead for missing optional fields
- **Repeated Field Optimization**: Efficient arrays and lists storage

4. Metadata Efficiency

- **Column Statistics**: Enable query optimization without reading data
- **Page-Level Metadata**: Fine-grained optimization opportunities
- **Schema Evolution**: Add columns without rewriting existing data

Compression Effectiveness Examples

Example 1: Categorical Data

```
Original CSV: 1GB
- Country column with 200 unique values in 10M rows
- Repeated string values consume significant space

Parquet with Dictionary Encoding: 150MB (85% reduction)
- Dictionary: 200 strings
- Data: 10M integers (4 bytes each)
```

Example 2: Time Series Data

```
Original CSV: 500MB
- Timestamp column with 1-second intervals
- Sensor readings with limited precision

Parquet with Delta + Bit-Packing: 75MB (85% reduction)
- Delta encoding for timestamps
- Bit-packing for sensor values
```

Example 3: Sparse Data

```
Original JSON: 2GB
- Many optional fields
- Nested structures with missing values

Parquet with Optimal Encoding: 200MB (90% reduction)
- No storage for missing optional fields
- Efficient nested structure representation
```

Storage Cost Implications

Cost Savings Analysis:

- **Storage Costs**: 70-90% reduction in storage requirements
- **Network Costs**: Proportional reduction in data transfer
- **Processing Costs**: Faster processing due to less I/O
- **Backup Costs**: Smaller backup sizes and faster backup/restore

---

Distributed File Management

Multi-Part File Structure

Why Files are Split into Parts

Technical Reasons:

1. **Distributed Processing Parallelism**
2. **HDFS Block Size Alignment**
3. **Memory Management Optimization**
4. **Fault Tolerance Enhancement**
5. **Load Balancing Improvement**

Part File Naming Convention

```
output_directory/
├── part-00000-<uuid>.snappy.parquet
├── part-00001-<uuid>.snappy.parquet
├── part-00002-<uuid>.snappy.parquet
└── _SUCCESS
```

Naming Components:

- **part-XXXXX**: Sequential part number
- **UUID**: Unique identifier preventing conflicts
- **Compression**: Indicates compression algorithm
- **Extension**: File format identifier

Distributed Processing Integration

Apache Spark Integration

Writing Parquet Files:

```
// Scala Example
df.write
  .mode("overwrite")
  .option("compression", "snappy")
  .parquet("hdfs://path/to/output")

// Multiple parts created automatically based on:
// - DataFrame partitioning
// - Executor configuration
// - Data size
```

Reading Parquet Files:

```
// Automatic part file discovery and reading
val df = spark.read.parquet("hdfs://path/to/output")

// Spark automatically:
// - Discovers all part files
// - Reads them in parallel
// - Creates unified DataFrame
```

Partitioning Strategies

1. Hash Partitioning:

```
df.repartition($"column_name")
  .write.parquet("output_path")
```

2. Range Partitioning:

```
df.repartitionByRange($"timestamp")
  .write.parquet("output_path")
```

3. Custom Partitioning:

```
df.repartition(numPartitions, $"key_column")
```

```
  .write.parquet("output_path")
```

File Size Optimization

Optimal Part File Sizes

Guidelines:

- **Target Size**: 64MB - 1GB per file
- **HDFS Block Size**: Align with HDFS block size (128MB or 256MB)
- **Memory Constraints**: Consider available executor memory
- **Query Patterns**: Balance between parallelism and overhead

Size Control Techniques:

```
// Control number of output files
df.coalesce(numFiles).write.parquet("output")

// Repartition for optimal size
df.repartition(calculateOptimalPartitions(dataSize))
  .write.parquet("output")
```

Fault Tolerance and Recovery

Failure Scenarios and Handling

1. **Node Failure During Write**:
- **Impact**: Only affected parts fail
- **Recovery**: Restart failed tasks only
- **Benefit**: Partial progress preserved

2. **Corruption in Part File**:
- **Impact**: Single part affected
- **Recovery**: Re-read from other replicas
- **Benefit**: Dataset remains accessible

3. **Network Partition**:
- **Impact**: Subset of parts inaccessible
- **Recovery**: Automatic failover to replicas
- **Benefit**: Continued processing possible

Query Optimization Features

Predicate Pushdown

Mechanism

Parquet stores column statistics at multiple levels:

- **File Level**: Overall min/max for entire file
- **Row Group Level**: Min/max for each row group
- **Page Level**: Min/max for each page

Query Optimization Process

1. **Parse Query Predicates**: Extract filter conditions
2. **Check File Statistics**: Skip entire files if possible
3. **Check Row Group Statistics**: Skip irrelevant row groups
4. **Check Page Statistics**: Skip irrelevant pages
5. **Read Minimal Data**: Process only necessary data

Example Optimization

```sql
SELECT customer_id, order_amount
FROM orders
WHERE order_date BETWEEN '2023-01-01' AND '2023-01-31'
```

Optimization Steps:

1. Check file-level `order_date` statistics
2. Skip files with no overlap with date range
3. Within relevant files, skip row groups outside date range
4. Read only `customer_id`, `order_amount`, `order_date` columns

Column Pruning

Mechanism

- **Schema Analysis**: Identify required columns from query
- **I/O Optimization**: Read only necessary column chunks
- **Memory Efficiency**: Load only required data into memory

Performance Impact

```
Query requiring 3 out of 20 columns:
- Row-based format: Must read entire rows (100% I/O)
- Parquet: Read only 3 columns (15% I/O)
- Performance improvement: 6.7x reduction in I/O
```

Data Skipping with Statistics

Column Statistics Stored

- **Min/Max Values**: For range-based filtering
- **Null Count**: For null-aware optimizations
- **Distinct Count**: For cardinality estimation
- **Total Count**: For aggregation pre-computation

Bloom Filters (Advanced Feature)

- **Purpose**: Efficient existence checks
- **Use Case**: Point lookups and IN clauses
- **Trade-off**: Small storage overhead for query speedup

Schema Evolution Support

Supported Evolution Types

1. **Adding Columns**:

```sql
-- Original Schema: id, name
-- New Schema: id, name, email (with default)
ALTER TABLE users ADD COLUMN email STRING DEFAULT 'unknown@domain.com'
```

2. **Removing Columns**:

```sql
-- Queries simply ignore removed columns
-- No data rewriting required
```

3. **Renaming Columns**:

```sql
-- Handled through schema mapping
-- Physical data unchanged
```

4. **Changing Data Types** (Limited):

```sql
-- Widening conversions supported
-- INT32 -> INT64 (supported)
-- STRING -> INT32 (not supported)
```

Performance Characteristics

Read Performance

Analytical Queries

- **Column Access**: 5-10x faster than row-based formats
- **Aggregations**: Vectorized processing advantages
- **Filtering**: Predicate pushdown eliminates unnecessary I/O

Scan Performance Factors

1. **Column Selectivity**: Fewer columns = better performance
2. **Row Group Size**: Optimal size for memory and parallelism
3. **Compression**: Balance between size and decompression cost
4. **Storage Medium**: SSD vs HDD performance characteristics

Write Performance

Trade-offs

- **Initial Write Cost**: Higher CPU usage for encoding/compression
- **Long-term Benefits**: Faster subsequent reads
- **Memory Usage**: Higher memory requirements during writing

Optimization Strategies

1. **Batch Size**: Larger batches improve compression efficiency
2. **Sorting**: Pre-sorting data improves encoding effectiveness
3. **Partitioning**: Optimal partitioning reduces write overhead

Memory Usage Patterns

Read Operations

- **Column Chunks**: Loaded into memory per column
- **Decompression**: Additional memory for decompressed data
- **Vectorization**: Memory for vectorized processing

Write Operations

- **Buffering**: Data buffered before writing to optimize compression
- **Dictionary Building**: Memory for dictionary construction
- **Metadata**: Memory for statistics collection

Best Practices and Implementation

Schema Design Guidelines

1. Column Ordering

```
Recommended Order:
1. Frequently filtered columns (for predicate pushdown)
2. Frequently selected columns (for column pruning)
3. Large/infrequently used columns (minimize I/O impact)
```

2. Data Types Selection

- **Use Appropriate Precision**: Don't use DOUBLE for integers
- **Consider Encoding**: Low-cardinality strings benefit from dictionary encoding
- **Nested Structures**: Use when natural, avoid over-nesting

3. Partitioning Strategy

```
// Partition by frequently filtered columns
df.write
  .partitionBy("year", "month")
  .parquet("data/orders")

// Results in directory structure:
// data/orders/year=2023/month=01/part-*.parquet
// data/orders/year=2023/month=02/part-*.parquet
```

Configuration Optimization

Spark Configuration

```
spark.conf.set("spark.sql.parquet.compression.codec", "snappy")
spark.conf.set("spark.sql.parquet.block.size", "268435456") // 256MB
spark.conf.set("spark.sql.parquet.page.size", "1048576")    // 1MB
spark.conf.set("spark.sql.parquet.enableVectorizedReader", "true")
```

Writing Optimization

```
df.repartition(numOptimalPartitions)
  .sortWithinPartitions("frequently_filtered_column")
  .write
  .option("compression", "snappy")
  .mode("overwrite")
  .parquet("output_path")
```

Monitoring and Troubleshooting

Key Metrics to Monitor

1. **File Size Distribution**: Avoid too many small files
2. **Compression Ratios**: Monitor encoding effectiveness
3. **Query Performance**: Track read/write performance
4. **Resource Usage**: Monitor CPU and memory usage

Common Issues and Solutions

Problem: Too many small files **Solution**: Use `coalesce()` or `repartition()` before writing
Problem: Poor compression ratios **Solution**: Analyze data characteristics and adjust encoding
Problem: Slow write performance **Solution**: Increase batch sizes and optimize partitioning
Problem: High memory usage during reads **Solution**: Adjust row group sizes and enable vectorized reading

Comparison with Other Formats

Parquet vs. CSV

Aspect	Parquet	CSV
Size	80-90% smaller	Baseline
Read Performance	5-10x faster for analytics	Slower
Write Performance	Slower initial write	Faster
Schema	Self-describing	External
Compression	Advanced	Basic
Tool Support	Big data tools	Universal

Parquet vs. ORC

Aspect	Parquet	ORC
Ecosystem	Spark, general big data	Hive-centric

Aspect	Parquet	ORC
Compression	Excellent	Excellent
Complex Types	Superior nested support	Good
ACID Support	Limited	Full (in Hive)
Schema Evolution	Good	Limited

Parquet vs. Avro

Aspect	Parquet	Avro
Storage Layout	Columnar	Row-based
Analytics	Optimized	General purpose
Schema Evolution	Good	Excellent
Streaming	Not ideal	Excellent
Size	Smaller	Larger

Summary and Key Takeaways

When to Use Parquet

Ideal Use Cases:

- Analytical and OLAP workloads
- Data warehousing and business intelligence
- Long-term data storage and archival
- Columnar aggregations and reporting
- Big data processing with Spark/Hive

Not Ideal For:

- Transactional (OLTP) workloads
- Frequent small updates
- Row-by-row processing
- Real-time streaming (primary format)

Critical Success Factors

1. **Understand Your Data**: Analyze characteristics for optimal configuration
2. **Design for Queries**: Organize data based on query patterns
3. **Monitor Performance**: Continuously optimize based on usage patterns
4. **Balance Trade-offs**: Consider write cost vs. read benefits
5. **Leverage Ecosystem**: Use tools optimized for Parquet

Future Considerations

- **Evolution of Standards**: Stay updated with Parquet format versions
- **Cloud Integration**: Leverage cloud-native optimizations
- **Hardware Trends**: Adapt to new storage and compute technologies
- **Ecosystem Development**: Monitor new tools and optimizations

Parquet represents a fundamental shift toward columnar storage in big data systems, offering significant advantages for analytical workloads through its sophisticated encoding, compression, and optimization techniques. Success with Parquet requires understanding both its technical capabilities and the specific requirements of your use cases.