# File format and engine object model transformation performance

## Abstract

Current Iceberg readers rely on direct conversion between the file format object model and the internal engine object model. This approach offers performance benefits by minimizing transformation overhead. However, it reduces maintainability: for 2 engines (or 3 object models, including vectorized paths) and 3 file formats, 9 separate transformations must be implemented.

During the discussion of the File Format API proposal [1] and its implementation [2], we decided to evaluate the implications of introducing an intermediate transformation layer to reduce the number of required conversion implementations.

## Goals

- Collect data to inform the decision on the best path forward.

## Non-goals

- We do not aim to define the future interface details at this stage.

## Test scenarios

We identified typical scenarios and simulated them using existing transformation paths. Our focus was on two main approaches:

- **Direct Readers**: Direct transformation from the file format object model to the engine-specific object model.
- **Arrow-Based Readers**: An intermediate Arrow-based layer is used for vectorized reading, followed by transformation to the engine-specific model.

Note: In many cases, readers are implemented as functions, and transformation costs are only incurred when columns are accessed. This was considered during testing.

### Environment

Tests were executed on a local development machine:

- Apple M4 Pro
- 48 GB RAM
- SSD
- macOS 15.5
- Openjdk 21.0.7 2025-04-15

Test code is available on this branch [3].

# Direct readers

In most cases, engines use their own object models, requiring a single transformation layer from the file format model. In some edge cases, engines operate directly on the file format model. For example:

- **Hive** uses ORC's VectorizedRowBatch directly in its vectorized object model, eliminating transformation costs. This is further optimized in Hive LLAP, where ORC ColumnVectors are cached in memory and on SSD in uncompressed form.
- **Trino, StarRocks**, and others also implement direct readers.

## Test case description

The tests are implemented in *SparkORCReadersFlatDataBenchmark*

1. Double Transformation: *readVectorized*
    - ORC read using VectorizedSparkOrcReaders → Spark ColumnarBatch (1st conversion)
    - Accessing Spark objects via accessors (2nd conversion)
2. No Transformation: *readVectorizedWithNoTransform*
    - ORC read using NoopBatchReader, returning VectorizedRowBatch directly
    - Data fields accessed to ensure evaluation

## Results

| Benchmark | Mode | Cnt | Score | Error | Units |
|---|---|---|---|---|---|
| SparkORCReadersFlatDataBenchmark .readVectorized | ss | 20 | 2.606 | ± 0.032 | s/op |
| SparkORCReadersFlatDataBenchmark .readVectorizedWithNoTransform | ss | 20 | 2.040 | ± 0.113 | s/op |

**Conclusion**: Double conversion introduces ~20% performance overhead.

# Intermediate arrow reader

We evaluated whether current single-line readers could be replaced with vectorized readers, using Arrow as an intermediate layer, followed by transformation to the engine-specific model.

## Spark arrow reader

### Test case description

The tests are implemented in *SparkVectorizedParquetReadersBenchmark*

1. Current Reader: *readCurrentReader*
    - Parquet read using SparkParquetReaders → InternalRow

2. Vectorized Reader with Transformation: *readVectorizedToInternalRow*
    - Parquet read using VectorizedSparkParquetReaders → ArrowBatchReader (1st conversion)
    - Conversion to InternalRow (2nd conversion)

#### Results

| Benchmark | Mode | Cnt | Score | Error | Units |
|---|---|---|---|---|---|
| SparkVectorizedParquetReadersBenchmark .readCurrentReader | ss | 20 | 4.165 | ± 0.020 | s/op |
| SparkVectorizedParquetReadersBenchmark .readVectorizedToInternalRow | ss | 20 | 2.616 | ± 0.016 | s/op |

**Conclusion**: Vectorized reading offers ~40% performance boost over single-line reading.

### Flink arrow reader

#### Test case description

The tests are implemented in *FlinkParquetReadersFlatDataBenchmark*

1. Current Reader: *readCurrentReader*
    - Parquet read using FlinkParquetReaders → RowData
2. Vectorized Reader with Transformation: *readVectorizedToRowData*
    - Parquet read using ArrowBatchReader (1st conversion)
    - Conversion to RowData (2nd conversion)

#### Results

| Benchmark | Mode | Cnt | Score | Error | Units |
|---|---|---|---|---|---|
| FlinkParquetReadersFlatDataBenchmark .readCurrentReader | ss | 20 | 3.778 | ± 0.031 | s/op |
| FlinkParquetReadersFlatDataBenchmark .readVectorizedToRowData | ss | 20 | 3.341 | ± 0.025 | s/op |

**Conclusion**: Vectorized reading offers ~40% performance gain over single-line reading.

# Arrow reader limitations

From the ArrowReader Javadoc [4], current limitations include:

- Complex types are not supported types
    - MapType
    - ListType
    - StructType
    - VariantType

- Type promotion not supported
- Constant values not supported
- Some primitive types are not supported:
    - FixedType
    - DecimalType

**Disclaimer**: Some of these might have been solved already, the others need to be fixed if we would like to use the Arrow readers everywhere.

# Summary

If we can close the functionality gap between direct and Arrow-based readers without significant performance loss, Arrow readers could replace single-line readers and offer a performance boost.

For Vectorized reading direct conversion remains superior and critical for performance-sensitive use cases.

# Conclusion

Direct Reading is ideal when:

- The file format and engine share a compatible object model.
- Performance is a critical concern.

Transformation Layers are preferable when:

- Supporting multiple file formats and engines.
- Prioritizing modularity, maintainability.
- The file format and engine have divergent data models.

We should follow a hybrid approach:

- Provide an Arrow-based transformation layer for interoperability. This would allow easy integration of engines and file formats where the performance is not the main goal.
- Allow engines to replace the transformation for tightly integrated formats. This will allow engines to maximize performance.

# References

[1] - File Format API proposal: https://docs.google.com/document/d/1sF_d4tFxJsZWsZFCyCL9ZE7YuI7-P3VrzMLIrrTIxds
[2] - File Format APi implementation: https://github.com/apache/iceberg/pull/12774#discussion_r2093626096
[3] - Conversion performance test branch: https://github.com/pvary/iceberg/tree/perf_bench
[4] - ArrowReader javadoc: https://github.com/apache/iceberg/blob/a69af4985fee4d6be428ba6f60c9e8f0c07da8fb/arrow/src/main/java/org/apache/iceberg/arrow/vectorized/ArrowReader.java#L85