# Iceberg DataFile reader and writer API proposal

## Abstract

Iceberg currently supports 3 different file formats: Avro, Parquet, ORC. With the introduction of Iceberg V3 specification many new features are added to Iceberg. Some of these features like new column types, default values require changes at the file format level. The changes are added by individual developers with different focus on the different file formats. As a result not all of the features are available for every supported file format.
Also there are emerging file formats like Vortex [1] or Lance [2] which either by specialization, or by applying newer research results could provide better alternatives for certain use-cases like random access for data, or storing ML models.

## Goals

The goal of this proposal is:

- Provide a clean, well defined API which file formats need to implement
- Implement the new API for the supported file formats
- Keep backward compatibility for the current readers/writers
- Simplify the existing code by removing the code duplications by using the common API instead of the big switch/case blocks
- Provide a test suite to validate the supported file format implementations

## Non goals

Implementing the missing features for the supported file formats could be done independently.
It is not a goal of this proposal to introduce new file formats. If the community decides so, that could be done independently later.

## Design

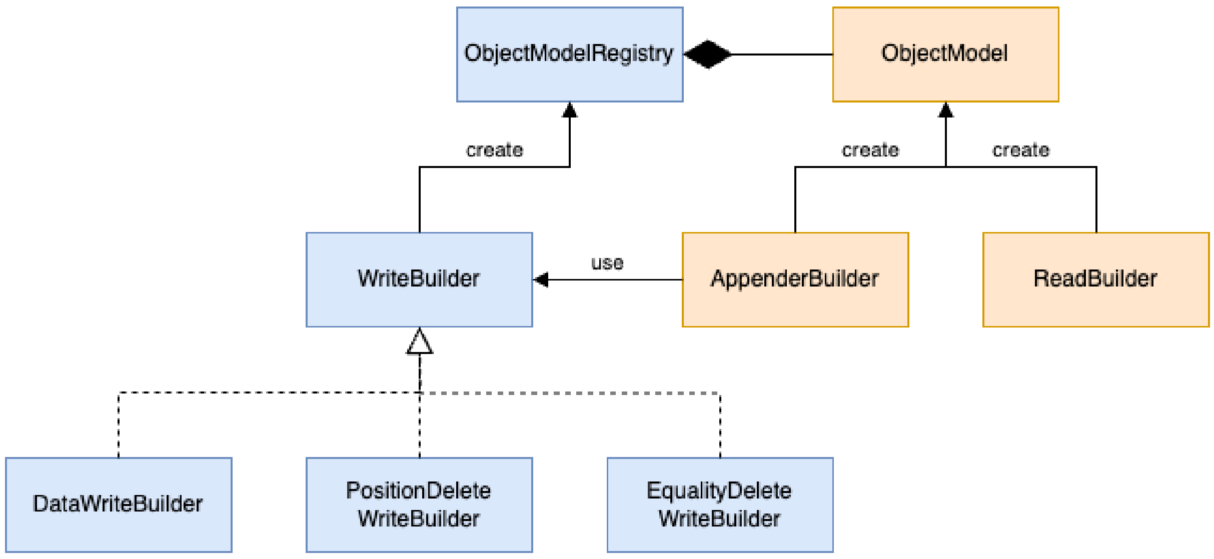
Each file format reads data into its own object model, and each query engine maintains its own object model. To avoid performance degradation, we continue using the current strategy: applying specific transformations between the engine's object model and the file format's object model.

- Reading: File format-specific objects are loaded into memory and then converted into the engine-specific object model.
- Writing: The engine-specific object model is transformed into the file format-specific object model.

This results in a matrix of transformations. To encapsulate these conversions, we define the Object Model API, which provides ReadBuilder and AppenderBuilder interfaces for reading and writing data.

Object models are registered in the Object Model Registry, which exposes ReadBuilder and AppenderBuilder instances to users. These builders can also be wrapped into writer-specific builders such as:

- DataWriteBuilder for data files
- PositionalDeleteWriteBuilder for positional delete files
- EqualityDeleteWriteBuilder for equality delete files



## Proposed changes in the iceberg-api module

### Object Model

To maintain performance, direct conversions are used between file formats and engine-internal formats. Object models encapsulate these conversions and must provide:

- ReadBuilder - for reading data files stored in a given File Format into the engine specific object model,
- AppenderBuilder - for writing engine specific object model to data/delete files stored in a given File Format.

Engines can implement and register their own object models to leverage Iceberg's read/write capabilities.

### Read Builder

File formats implement this interface to provide a builder for reading data files. The builder is parameterized by user input and returns objects defined by the associated object model. This interface is directly exposed to users for reader customization.

### Appender Builder

File formats implement this interface to provide a builder for writing. It is configured using user-provided parameters, and the build method generates the appropriate appender.

The appender must support:

- Engine-specific input types for data and equality delete modes
- Positional deletes, where the row type is PositionDelete

This interface is directly exposed to users for appender customization or wrapped for use in content file writers.

## Proposed changes in the iceberg-data module

### Object Model Registry

The available Object Models are registered by the registerObjectModel method. These Object Models will be used to create the Read Builders and the Appender Builders. The former ones are returned directly, the later ones either used directly, or wrapped into the appropriate writer builder implementations.

### Data Write Builder

Interface which defines the methods that are needed for Data File write builders. The Data File Writer will expect inputs defined by the engineSchema which should be convertible to the Iceberg schema defined.

### Position Delete Write Builder

Interface which defines the methods that are needed for Position Delete File write builders. The Position Delete File Writer will expect Position Delete records. If the row schema is set then the positional delete records should contain the deleted rows specified by the engineSchema. The provided engine schema should be convertible to the Iceberg schema.

### Equality Delete Write Builder

Interface which defines the methods that are needed for Equality Delete File write builders. The Equality Delete File Writer will expect inputs defined by the engineSchema which should be convertible to the Iceberg schema.

### Write Builder

Implementation for the different Write Builder interfaces. The builder is an internal class and could change without notice. The users should use one of the following specific interfaces instead:

- Data Write Builder
- Position Delete Write Builder
- Equality Delete Write Builder

The Write Builder wraps the file format specific Appender Builder. To allow further engine and file format specific configuration changes for the given writer, the Appender Builder's build method is called to create the appender which is used by the Write Builder to provide the required functionality.

## Packaging structure

### File Format API

- These interfaces are implemented by supported file format implementations.
- Includes ObjectModel, AppenderBuilder, and ReadBuilder.
- Located in the Iceberg Core project, because the NameMapping, MetricsConfig classes are required on the interfaces and part of the core project.

### File Write API

- Used by engines to register object models and retrieve the appropriate builders for reading/writing.
- Includes ObjectModelRegistry, write builder interfaces, and their implementations.
- Located in the Iceberg Data project.

## Code duplications

Currently, file format-specific logic is implemented using large switch statements, leading to duplicated code. After refactoring:

- These blocks will be replaced with a single call to fetch the correct builder from the ObjectModelRegistry.
- Each file format will no longer need to implement its own data and delete write builders.
    - Format-specific logic will move into the object model.
    - Duplicated code will be deprecated in favor of the unified File Write API.

# Test suite

We should organize the test suites to have separate tests for the different table spec versions.

The test suite should test all the features which are expected from a file format.

- Builder properties:
    - Split handling
    - Projection
    - Case sensitivity
    - Filtering
    - Container reuse
    - Batch size
- Reading and writing supported data types:
    - BOOLEAN

    - INTEGER
    - LONG
    - FLOAT
    - DOUBLE
    - DATE
    - TIME
    - TIMESTAMP
    - STRING
    - UUID
    - FIXED
    - BINARY
    - DECIMAL
    - STRUCT
    - LIST
    - MAP
    - TIMESTAMP_NANO - v3 only
    - VARIANT - v3 only
    - UNKNOWN - v3 only
    - Geometry - v3 only
- Returning metadata columns:
    - FILE_PATH
    - ROW_POSITION
    - IS_DELETED
    - SPEC_ID
    - PARTITION_COLUMN
        - Transformations
        - Partition evolution (adding and removing columns)
- Schema evolution:
    - Adding column (reading with wider schema)
    - Projection/Removing column (reading with narrower schema)
    - Removing and adding a column with the same name (name mapping)
    - Allowed type changes
    - Reorder columns
- Metrics collection
- Default values - v3 only
- Delete filter handling - v3 only?
    - Removed rows
    - Counter handling
- Encryption - v3 only

## Design Alternatives

There are several alternatives which were discussed and rejected during the evaluation, but for future reference they are kept below.

# Minimal changes

## Registry

Similarly to the InternalData read and write builders proposal [3], we can introduce a registry for the data file readers and writers. The registry would store/and return the readers/writer factories based on the following selection criteria:

1. Data file format
2. Row data type (input for readers, output for writers)
3. Builder type - needed when we have multiple available implementations, like Parquet Comet/Iceberg readers

## Builder

Current file format builders (ReadBuilder/WriteBuilder/DataWriteBuilder/DeleteWriteBuilder) are very similar for the supported file formats. There are only minimal differences which we can deprecate and consolidate. We should create a common interface which will be part of the new API. These builders should implement this interface. The Builders contain the file format specific codes.

## DeleteFilter

We need to formalize the DeleteFilter API as it is pushed down to the Parquet vectorized reader to filter out the records on the reader side.

## BuilderFactory

The following builder factory groups are planned for the different row data types

- Record - Generic reader/writer
- ColumnarBatch (Arrow) - for the Arrow reader
- RowData - for the Flink reader/writer
- InternalRow - for the Spark reader/writer
- ColumnarBatch (Spark) - for the Spark vectorized reader

The BuilderFactory contains the conversion specific codes which converts:

- In case of the readers the file format specific raw data to the target type
- In case of the writers the target type to the file format specific raw data

We need to create a matrix of builder factories for every supported row data type. This guarantees performance since we can avoid an extra conversion to a common type and convert everything directly to the target type.

The conversion code (former createReaderFunc/createBatchedReaderFunc) needs the following parameters:

- In case of the readers:
    - InputFile to read
    - Task to provide the values for metadata columns (_file_path, _spec_id, _partition)
    - ReadSchema to use when reading the data file

    - PartitionType (based on all specs in the table) which is used for calculating values for _partition column
    - DeleteFilter is used when the delete record filtering is pushed down to the reader
- In case of the writers:
    - OutputFile to write
    - Input data schema

## Fewer interfaces

There are many duplicated properties for the different builders. If we use inheritance to remove the duplication then we will end up with a high number of interfaces and base classes. We could accept the code duplication and create a single interface/base class for:

- ReaderBuilder
- AppenderBuilder
- DataWriterBuilder
- We still need to use inheritance for EqualityDeleteWriterBuilder and PositionalDeleteWriterBuilder to easily migrate the current Avro/Parquet/ORC implementations.

See: https://github.com/pvary/iceberg/tree/file_format_api_minimal_few_class

## Simplify builder API

In all of the pervious alternatives the builder API required information which needed to for the parametrization of reader transformations:

```
ReaderBuilder<?> builder(
    InputFile inputFile,
    ContentScanTask<?> task,
    Schema readSchema,
    Table table,
    DeleteFilter<?> deleteFilter);
```

Some of it is duplicated information already configured for the builder:

```
T split(long newStart, long newLength);
T project(Schema newSchema);
T filter(Expression newFilter);
```

We could allow the ReaderService to initialize the builder before the actual build method is called. This could enable the service to use all of the data provided by the builder parametrization, and result in a cleaner interface.

See: https://github.com/pvary/iceberg/tree/file_format_api_builder_only

## Interface only API

In the previous alternatives the builder API provided base classes which handled the common build parameters. While that approach removed duplicated boilerplate code for builders it restricted implementation inheritance and increased the size of the changeset. Alternatively, we can provide an interface only API and leave the boilerplate code in the format specific classes (Parquet.java/Avro.java/ORC.java)

See: https://github.com/apache/iceberg/pull/12298

[1] https://github.com/spiraldb/vortex
[2] https://lancedb.github.io/lance/
[3] https://github.com/apache/iceberg/pull/12060 - Core: Add InternalData read and write builders
[4] https://github.com/apache/iceberg/pull/12069 - WIP generic reader
[5] https://github.com/apache/iceberg/pull/12164 - WIP File format write