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 TCK to validate the supported file format implementations

Non goals

The TCK will help identify the missing features for the supported file formats, but 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.
It is not a goal of this proposal to change the internals of the current file formats.
Puffin readers and writers (like the ones used by DVs) are out of scope for this proposal.

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.

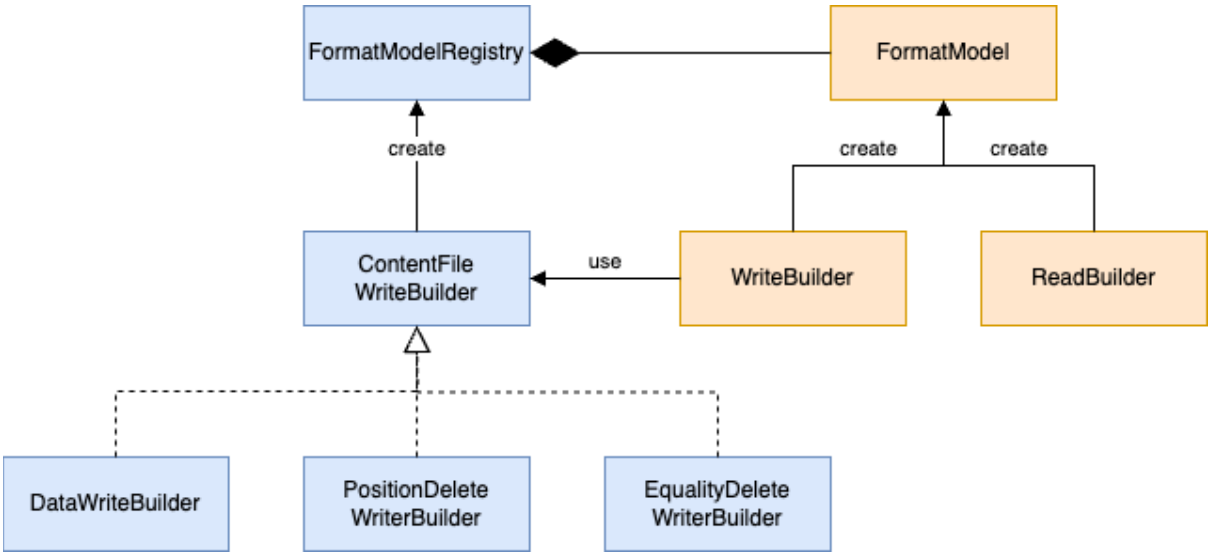
- Reading: File format-specific objects are loaded into memory and then converted into the engine-specific object model.

- 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 Format Model API, which provides ReadBuilder and WriteBuilder interfaces for reading and writing data.

Format models are registered in the Format Model Registry, which exposes ReadBuilder and WriteBuilder 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 implemented for every File Format

Format 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,
- WriteBuilder - 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.

The interface will not have generic parameters, so the existing format specific implementations (Parquet.ReadBuilder, Avro.ReadBuilder, ORC.ReadBuilder) could simply extend the new interface and

Write 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. Position deletes are written using a specific PositionDelete Object model

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

The interface will not have generic parameters, so the existing format specific implementations (Parquet.WriteBuilder, Avro.WriteBuilder, ORC.WriteBuilder) could simply extend the new interface and

Method names

I have collected the method names from the old API, and the proposed new API.
We should go through them and decide what should be the new naming.
Builder methods

Proposed changes in the iceberg-core module

Format Model Registry

The available Format Models are registered by the register method. These Format Models will be used to create the Read Builders and the Write Builders. The former ones are returned directly, the later ones are wrapped into the appropriate writer builder implementations.
Re-registering Format Models is not allowed as a general rule. To prevent issues where classes are reloaded by Flink and Spark, the registered model is updated if the classname, the file format, the object type and the schema type are the same.

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. Setting the row schema is not supported.

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.

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

Exposing Variants

The proposal is to expose *WriteBuilder.inputSchema* and *ReadBuilder.outputSchema*, which can guide the shredding of variant columns. These schemas would define the expected structure, including both the shredded columns and a dedicated variant column to capture any non-shredded data.
This schema-driven approach enables compatibility across different engine-specific representations. For example, if a DataFrame stores values as *tinyint* or *shortint*, the writer could convert and persist them as *integer* columns.

Packaging structure

The NameMapping, MetricsConfig classes are required on the interfaces and part of the core project, so the new files are located in the Iceberg Core project in the *org.apache.iceberg.formats* package.

File Format API

- These interfaces are implemented by supported file format implementations.
- Includes FormatModel, WriteBuilder, and ReadBuilder.

File Write API

- Used by engines to register object models and retrieve the appropriate builders for reading/writing.
- Includes FormatModelRegistry, write builder interfaces, and their implementations.

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 FormatModelRegistry.
- Each file format will no longer need to implement its own data and delete write builders.
    - Format-specific logic will move into the format model.
    - Duplicated code will be deprecated in favor of the unified File Write API.

Backwards compatibility

We need to keep the current file format specific WriteBuilders and ReadBuilders minimally until the deprecation period. During this period we can reuse the old WriteBuilders and ReadBuilders to implement the new API. This allows us to keep the old API intact. After the removal of the old API the implementation needs to be moved, but with the help of the TCK we can ensure that the functionality remains the same.

TCK

Unit tests

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

Open Questions

Expose Parquet.ReaderFunction

While implementing the InternalData interfaces for Parquet, Russell came up with the idea to replace the 3-4 reader functions with a single ReaderFunction.
See:

- PR: https://github.com/apache/iceberg/pull/14040
- ReaderFunction: https://github.com/apache/iceberg/blob/042f01a240439691ac67c6d67bd51cf5a4167dd2/parquet/src/main/java/org/apache/iceberg/parquet/Parquet.java#L1171

We should follow the same pattern for every ReadBuilder/WriteBuilder (Parquet, Avro, ORC) internally. This could be done in a parallel PR before finalizing this PR.

The implementation of the new FormatModel will also require us to create more ReaderFunction/WriteFunction interfaces, and we could utilize the changes mentioned above.
See:

- https://github.com/apache/iceberg/pull/12298/files#diff-f96b8e4567069fcc597f1802541029d2d1707ec5bc128a631ed7244bf364e22bR84-R107

```
@FunctionalInterface
public interface ReaderFunction<D> {
 ParquetValueReader<D> read(
     Schema schema, MessageType messageType, Map<Integer, ?>
constantFieldAccessors);
}

@FunctionalInterface
public interface BatchReaderFunction<D, F> {
 VectorizedReader<D> read(
     Schema schema,
     MessageType messageType,
     Map<Integer, ?> constantFieldAccessors,
     F deleteFilter,
     Map<String, String> config);
}

@FunctionalInterface
public interface WriterFunction<S> {
 ParquetValueWriter<?> write(Schema icebergSchema, MessageType messageType, S
engineSchema);
}
```

Instead of these functional interfaces, we could possibly expose these functions on the FormatModel constructors, like:

```
private ParquetFormatModel(
   Class<D> type,
   Class<S> schemaType,
   Parquet.ReadBuilder.ReaderFunction readerFunction,
   Parquet.ReadBuilder.BatchReaderFunction batchReaderFunction,
   Parquet.WriteBuilder.WriterFunction writerFunction) {
```

The registration of the Format Models will change from this:

```
FormatModelRegistry.register(
   new ParquetFormatModel<>(
       Record.class,
       Schema.class,
       GenericParquetReaders::buildReader,
       (schema, messageType, inputType) ->
           GenericParquetWriter.create(schema, messageType))));
```

To this:

```
FormatModelRegistry.register(
   new ParquetFormatModel<>(
           Record.class,
```

```
            Schema.class,
            new Parquet.ReadBuilder.ReaderFunction() {
                @Override
                public Function<MessageType, ParquetValueReader<?>> apply() {
                    return messageType ->
GenericParquetReaders.buildReader(schema(), messageType);
                }
            },
            [..])));
```

This registration is a bit more chatty, but we can reuse the existing classes instead of creating new interfaces.

Revisit Generics in the API Design

There was a debate about whether the new ReadBuilder and WriteBuilder interfaces should use generic parameters. I would like to revisit it, as generics could provide stronger type safety by ensuring that builders accept the correct input types and produce the correct output types, especially when these builders are passed around as parameters.

The main concern raised by the community is that introducing generics would prevent reusing the current classes, like *Parquet.WriteBuilder* and *Parquet.ReadBuilder,* without significant changes, increasing the size and complexity of the changes.

Proposed compromise
Keep generic parameters in the new interfaces, but allow existing implementations to use Object as a fallback type. For example:

```
public interface WriteBuilder<D, S> {
  WriteBuilder<D, S> inputSchema(Object schema);
  FileAppender<D> appender() throws IOException;
}
```

Make the old WriteBuilder/ReadBuilder classes implement them with Object parameters:

```
public static class WriteBuilder
   implements org.apache.iceberg.formats.WriteBuilder<Object, Object> {

  public WriteBuilder inputSchema(Object newInputSchema) {
    Preconditions.checkNotNull(
      inputSchemaClass, "Input schema class must be set");
    Preconditions.checkArgument(
      inputSchemaClass.isInstance(newInputSchema),
      "Input schema must be of class: %s, found: %s",
      inputSchemaClass.getName(),
      newInputSchema.getClass().getName());
    this.inputSchema = newInputSchema;
    return this;
  }

  public FileAppender<Object> appender() throws IOException {
    return build();
  }
```

```
}
```

The FormatModel could then safely cast to the correct types internally:

```
public WriteBuilder<D, S> writeBuilder(OutputFile outputFile) {
 return (WriteBuilder<D, S>) Parquet.write(outputFile)
     .inputSchemaClass(schemaType)
     .writerFunction((WriterFunction<Object>) writerFunction);
}
```

This approach preserves type safety for new implementations while minimizing disruption for existing ones.

Options going forward:

1. **Keep both generic parameters (input/output type and schema type)**
    - Pro: Full type safety for both schema and data types.
    - Con: Requires introducing new method names to avoid clashes (e.g., build() vs. appender()).
2. **Keep only the schema type as a generic parameter**
    - Pro: At least schema type safety is preserved; simpler migration.
    - Con: Input/output type safety is lost, leading to partial consistency.
3. **Remove generics entirely**
    - Pro: Simplest migration path.
    - Con: Loses all compile-time type safety, making the API less robust.

References

[1] https://github.com/spiraldb/vortex
[2] https://lancedb.github.io/lance/
[3] https://github.com/apache/iceberg/pull/12774 - Core, Data: File Format API interfaces
[4] https://github.com/apache/iceberg/pull/12298 - Core: Interface based DataFile reader and writer API - PoC