FLIP-95: New TableSource and TableSink interfaces

Discussion thread	https://lists.apache.org/thread/xp87713sdky9xz51hb2ltt7c4bl3fvxw
Vote thread	
JIRA	FLINK-16987 - FLIP-95: Add new table source and sink interfaces OPEN
Release	

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

- Motivation
- Public Interfaces
  - Source Interfaces
    - DynamicTableSource
    - ScanTableSource
    - SupportsComputedColumnPushDown
    - SupportsWatermarkPushDown
    - SupportsFilterPushDown, SupportsProjectionPushDown
    - LookupTableSource
  - Sink Interfaces
  - Factory Interfaces
    - Factory
    - DynamicTableFactory, DynamicTableSourceFactory, DynamicTableSinkFactory
    - FormatFactory, DeserializationFormatFactory
  - Data Structure Interfaces
    - RowData
    - ArrayData
    - MapData
    - StringData
    - DecimalData
    - TimestampData
    - RawValueData
    - GenericRowData
    - GenericArrayData
    - GenericMapData
- Compatibility, Deprecation, and Migration Plan
- Test Plan
- Rejected Alternatives
  - Not moving Blink's data structures to `table-common`
  - Discussions around filter pushdown/projection pushdown
  - Other discussions

Motivation

The table sources and sink interfaces have grown over time from pure batch to streaming interfaces. More and more abilities were added which makes the current class structure already quite complex:

Current source interfaces:

- ProjectableTableSource<T>
- LookupableTableSource extends TableSource
- LimitableTableSource<T>
- FilterableTableSource<T>
- PartitionableTableSource
- ProjectableTableSource<T>
- NestedFieldsProjectableTableSource<T>
- StreamTableSource<T> extends TableSource<T>

Current sink interfaces:

- OverwritableTableSink
- PartitionableTableSink
- UpsertStreamTableSink<T> extends StreamTableSink<Tuple2<Boolean, T>>
- RetractStreamTableSink<T> extends StreamTableSink<Tuple2<Boolean, T>>
- StreamTableSink<T> extends TableSink<T>
- AppendStreamTableSink<T> extends StreamTableSink<T>
- BatchTableSink<T> extends TableSink<T>

Proper support for handling changelogs, more efficient processing of data through the new Blink planner, and unified interfaces that are DataStream API agnostic add further requirements.

The goals of this FLIP are:

- **Simplify the current interface architecture**:
  - Merge upsert, retract, and append sinks.
  - Unify batch and streaming sources.
  - Unify batch and streaming sinks.
- **Allow sources to produce a changelog**:
  - UpsertTableSources have been requested a lot by users. Now is the time to open the internal planner capabilities via the new interfaces.
  - According to FLIP-105, we would like to support changelogs for processing formats such as Debezium.
- **Don't rely on DataStream API for source and sinks**:
  - According to FLIP-32, the Table API and SQL should be independent of the DataStream API which is why the `table-common` module has no dependencies on `flink-streaming-java`.
  - Source and sink implementations should only depend on the `table-common` module after FLIP-27.
  - Until FLIP-27 is ready, we still put most of the interfaces in `table-common` and strictly separate interfaces that communicate with a planner and actual runtime reader/writers.
- **Implement efficient sources and sinks without planner dependencies**:
  - Make Blink's internal data structures available to connectors.
  - Introduce stable interfaces for data structures that can be marked as `@PublicEvolving`.

Public Interfaces

We propose the following interface structure.

Main interfaces:

- DynamicTableSource
- ScanTableSource extends DynamicTableSource
- LookupTableSource extends DynamicTableSource
- DynamicTableSink

Corresponding factory interfaces:

- Factory
- DynamicTableFactory extends Factory
- DynamicTableSourceFactory extends DynamicTableFactory
- DynamicTableSinkFactory extends DynamicTableFactory
- FormatFactory extends Factory

Optional interfaces that add further abilities:

- SupportsComputedColumnPushDown
- SupportsFilterPushDown
- SupportsProjectionPushDown
- SupportsWatermarkPushDown
- SupportsLimitPushDown
- SupportsPartitionPushDown
- SupportsPartitioning
- SupportsOverwriting

Data structure interfaces and classes:

- RowData
- ArrayData
- MapData
- StringData
- DecimalData
- TimestampData
- RawValueData
- GenericRowData implements RowData
- GenericArrayData implements ArrayData
- GenericMapData implements MapData

Main interfaces, factory interfaces, some ability interfaces, and data structure interfaces will be discussed below. Other abilities remain unchanged.

Source Interfaces

DynamicTableSource

```
/**
 * Source of a dynamic table from an external storage system.
 *
 * <p>A dynamic table source can be seen as a factory that produces concrete runtime implementation.
 *
 * <p>Depending on the optionally declared interfaces such as {@link SupportsComputedColumnPushDown} or
 * {@link SupportsProjectionPushDown}, the planner might apply changes to instances of this class and thus
 * mutates the produced runtime implementation.
 */
@PublicEvolving
public interface DynamicTableSource {

        /**
         * Returns a string that summarizes this source for printing to a console or log.
         */
        String asSummaryString();

        /**
         * Creates a copy of this instance during planning.
         */
        DynamicTableSource copy();

        // --------------------------------------------------------------------------------------------
        // Helper Interfaces
        // --------------------------------------------------------------------------------------------

        /**
         * Converts data structures during runtime.
         */
        interface DataStructureConverter extends RuntimeConverter {

                /**
                 * Converts the given object into an internal data structure.
                 */
                @Nullable Object toInternal(@Nullable Object externalStructure);
        }
}
```

Side note: `RuntimeConverter` is the general interface for converters for providing an `open()` method. This method can be used for triggering the compilation of code generated converters in the future. Because `SerializationSchema` and `DeserializationSchema` don't provide `open()` methods yet, we will need a separate discussion for introducing them. The `Context` is reserved for future use. We could e.g. allow access to MemorySegments or further low-level runtime access.

```
public interface RuntimeConverter extends Serializable {

        /**
         * Initializes the converter during runtime.
         *
         * <p>This should be called in the {@code open()} method of a runtime class.
         */
        void open(Context context);

        /**
         * Context for conversions during runtime.
         */
        interface Context {

                static Context empty() {
                        return new Context() {
                                // nothing to do
                        };
                }
        }
}
```

ScanTableSource

`ScanTableSource` is the main interface for both batch and streaming sources. For both insert-only and updating changelogs. Thus, it replaces `AppendTableSource` and exposes upsert, retract, and further functionality.

Compared to the previous design, we assume that the sources emit the Blink planner's data structures. This avoids conversion overhead during runtime. Furthermore, it allows to push down concepts such as computed columns and watermark assignment deep into the source (right next to the source's partitions).

The actual runtime implementation is created lazily in `getScanRuntimeProvider()`. If a connector takes a format (e.g. JSON for Kafka), this method will perform format discovery using the provided classloader.

The planner provides helper utilities for creating type information for Flink's data structures and converters such that user-code must not deal with Flink's data structures manually. For example, if the user has a POJO during runtime that needs to be converted into either a structured type or nested rows, a converter can be created for dealing with this conversion automatically (maybe even code generated).

The produced data type can be retrieved via `CatalogTable.getSchema().toProducedDataType()`.

Side note: Because there is no stable interface that is equivalent to `RowData` (previous `BaseRow`) for encoding changes, we propose to enrich `org.apache.flink.types.Row` with an additional `RowKind`. This allows users to pass `Row` instances to converters and get fully defined `RowData`s from the framework. Also, having a change flag in `Row` allows us to use it better in DataStream API bridging methods `TableEnvironment.toDataStream(table, ChangelogMode): DataStream<Row>` and result retrieval such as `table.collect(ChangelogMode): Iterable<Row>` in the future. Both kinds of methods would use `ChangelogMode` in a similar way as sources and sinks.

```
/**
 * A {@link DynamicTableSource} that scans all rows from an external storage system.
 *
 * <p>Depending on the specified {@link ChangelogMode}, the scanned rows must not contain only
 * insertions but can also contain updates and deletions.
 */
@PublicEvolving
public interface ScanTableSource extends DynamicTableSource {

        /**
         * Returns what kind of changes are produced by this source.
         *
         * @see RowKind
         */
        ChangelogMode getChangelogMode();

        /**
         * Returns the actual implementation for reading the data.
         */
        ScanRuntimeProvider getScanRuntimeProvider(Context context);

        // --------------------------------------------------------------------------------------------
        // Helper Interfaces
        // --------------------------------------------------------------------------------------------

        interface Context {

                /**
                 * Creates type information describing the internal data structures of the given
                 * {@link DataType}.
                 */
                TypeInformation<?> createTypeInformation(DataType producedDataType);

                /**
                 * Creates a runtime data structure converter that converts data of the given {@link DataType}
                 * to Flink's internal data structures.
                 *
                 * <p>Allows to implement runtime logic without depending on Flink's internal structures for
                 * timestamps, decimals, and structured types.
                 *
                 * @see LogicalType#supportsInputConversion(Class)
                 */
                DataStructureConverter createDataStructureConverter(DataType producedDataType);
        }

        /**
         * Actual implementation for reading the data.
         */
        interface ScanRuntimeProvider {

                /**
                 * Whether the data is bounded or not.
                 */
                boolean isBounded();
        }
}
```

`ScanRuntimeProvider` abstracts runtime interfaces. Which means we can support old interfaces such as `SourceFunction` and `InputFormat` and the new FLIP-27 sources in the future.

Since FLIP-27 sources will return whether they are bounded or not, the `ScanRuntimeProvider` interface already offers returning this property via `isBounded()`. So far this property is not required during logical optimization but only during physical optimization, which is why this property is not part of the top-level `ScanTableSource` class.

```
/**
 * Uses a {@link SourceFunction} during runtime for reading.
 */
interface SourceFunctionProvider extends ScanTableSource.ScanRuntimeProvider {

        SourceFunction<RowData> createSourceFunction();

        static SourceFunctionProvider of(SourceFunction<RowData> sourceFunction, boolean isBounded) {
                return new SourceFunctionProvider() {
                        @Override
                        public SourceFunction<RowData> createSourceFunction() {
                                return sourceFunction;
                        }

                        @Override
                        public boolean isBounded() {
                                return isBounded;
                        }
                };
        }
}

/**
 * Uses an {@link InputFormat} during runtime for reading.
 */
public interface InputFormatProvider extends ScanTableSource.ScanRuntimeProvider {

        InputFormat<RowData, ?> createInputFormat();

        static InputFormatProvider of(InputFormat<RowData, ?> inputFormat) {
                return new InputFormatProvider() {
                        @Override
                        public InputFormat<RowData, ?> createInputFormat() {
                                return inputFormat;
                        }

                        @Override
                        public boolean isBounded() {
                                return true;
                        }
                };
        }
}
```

`ChangelogMode` and `RowKind` define the set of changes in the data.

```
/**
 * A kind of row in a changelog.
 */
@PublicEvolving
public enum RowKind {

        /**
         * Insertion operation.
         */
        INSERT,

        /**
         * Previous content of an updated row.
         */
        UPDATE_BEFORE,

        /**
         * New content of an updated row.
         */
        UPDATE_AFTER,

        /**
         * Deletion operation.
         */
        DELETE
}
```

```
/**
 * The set of changes a {@link DynamicTableSource} produces or {@link DynamicTableSink} consumes.
 */
@PublicEvolving
public final class ChangelogMode {

        private final Set<RowKind> kinds;

        private ChangelogMode(Set<RowKind> kinds) {
                Preconditions.checkArgument(
                        kinds.size() > 0,
                        "At least one kind of row should be contained in a changelog.");
                this.kinds = Collections.unmodifiableSet(kinds);
        }

        public static Builder newBuilder() {
                return new Builder();
        }

        public Set<RowKind> getContainedKinds() {
                return kinds;
        }

        public boolean contains(RowKind kind) {
                return kinds.contains(kind);
        }

        public boolean containsOnly(RowKind kind) {
                return kinds.size() == 1 && kinds.contains(kind);
        }

        // --------------------------------------------------------------------------------------------

        public static class Builder {

                private final Set<RowKind> kinds = EnumSet.noneOf(RowKind.class);

                public Builder() {
                        // default constructor to allow a fluent definition
                }

                public Builder addContainedKind(RowKind kind) {
                        this.kinds.add(kind);
                        return this;
                }

                public ChangelogMode build() {
                        return new ChangelogMode(kinds);
                }
        }
}
```

SupportsComputedColumnPushDown

In theory, this interface could have been deeply integrated into `ScanTableSource`. However, we decided to go for modularization instead. We provide a `MapFunction`-like converter for enriching the produced data with computed columns. The converter might be code generated and works on `RowData`.

```
/**
 * Allows to push down computed columns into a {@link ScanTableSource}.
 */
@PublicEvolving
public interface SupportsComputedColumnPushDown {

        /**
         * Provides a converter that converts the produced {@link RowData} containing the physical
         * fields of the external system into a new {@link RowData} with push-downed computed columns.
         *
         * <p>For example, in case of
         * {@code CREATE TABLE t (s STRING, ts AS TO_TIMESTAMP(str), i INT, i2 AS i + 1)},
         * the converter will convert a {@code RowData(s, i)} to {@code RowData(s, ts, i, i2)}.
         *
         * <p>Use {@link TableSchema#toRowDataType()} instead of {@link TableSchema#toProducedRowDataType()}
         * for describing the final output data type when creating {@link TypeInformation}.
         */
        void applyComputedColumn(ComputedColumnConverter converter);

        /**
         * Generates and adds computed columns to a {@link RowData} if necessary.
         */
        interface ComputedColumnConverter extends RuntimeConverter {

                /**
                 * Generates and adds computed columns to a {@link RowData} if necessary.
                 */
                RowData convert(RowData row);
        }
}
```

SupportsWatermarkPushDown

Both `SupportsComputedColumnPushDown` and `SupportsWatermarkPushDown` can exist without each other. It should be the responsibility of the planner to ensure that both are implemented if necessary.

```
/**
 * Allows to push down watermarks into a {@link ScanTableSource}.
 */
@PublicEvolving
public interface SupportsWatermarkPushDown {

        void applyWatermark(WatermarkAssignerProvider assigner);

        // ------------------------------------------------------------------------------------------

        class WatermarkAssignerProvider {
                // marker interface that will be filled with FLIP-27
        }
}

// in table-api-java-bridge
/**
 * Uses a {@link AssignerWithPeriodicWatermarks} during runtime for watermarks.
 */
@PublicEvolving
public class PeriodicAssignerProvider extends SupportsWatermarkPushDown.WatermarkAssignerProvider {

        private AssignerWithPeriodicWatermarks<ChangelogRow> periodicAssigner;

        public PeriodicAssignerProvider(AssignerWithPeriodicWatermarks<ChangelogRow> periodicAssigner) {
                this.periodicAssigner = periodicAssigner;
        }

        public AssignerWithPeriodicWatermarks<ChangelogRow> getPeriodicAssigner() {
                return periodicAssigner;
        }
}

// in table-api-java-bridge
/**
 * Uses a {@link AssignerWithPunctuatedWatermarks} during runtime for watermarks.
 */
@PublicEvolving
public class PunctuatedAssignerProvider extends SupportsWatermarkPushDown.WatermarkAssignerProvider {

        private AssignerWithPunctuatedWatermarks<ChangelogRow> punctuatedAssigner;

        public PunctuatedAssignerProvider(AssignerWithPunctuatedWatermarks<ChangelogRow> punctuatedAssigner) {
                this.punctuatedAssigner = punctuatedAssigner;
        }

        public AssignerWithPunctuatedWatermarks<ChangelogRow> getPunctuatedAssigner() {
                return punctuatedAssigner;
        }
}
```

SupportsFilterPushDown, SupportsProjectionPushDown

Some examples of other important ability interfaces are shown in this section. They have been updated to satisfy the needs of various connectors where a `DynamicTableSource` needs to communicate with the planner.

In case of a filter push down, the optimizer needs to know which predicates are remaining. If there are remaining predicates, the optimizer will create a new filter node.

Furthermore, the following applies:

all predicates != pushed predicates + remaining predicates

For example, for Parquet all pushed predicates should be retained because a filter is applied on a group instead of each record.

In case of projection push down, the optimizer also needs to know which columns are remaining.

Because push down's modify the state of the table source and different push down's might interfere with each other, `DynamicTableSource` has a `copy()` method that will be used during planning.

There was a discussion whether we should check for supported/remaining predicates before pushing them down. However, it would require implementing a lot of methods in sources with similar logic.

```
/**
 * Allows to push down filters into a {@link ScanTableSource}.
 */
@PublicEvolving
public interface SupportsFilterPushDown {

        Result applyFilters(List<ResolvedExpression> filters);

        final class Result {
                private final List<ResolvedExpression> acceptedFilters;
                private final List<ResolvedExpression> remainingFilters;

                protected Result(
                                List<ResolvedExpression> acceptedFilters,
                                List<ResolvedExpression> remainingFilters) {
                        this.acceptedFilters = acceptedFilters;
                        this.remainingFilters = remainingFilters;
                }

                public List<ResolvedExpression> getAcceptedFilters() {
                        return acceptedFilters;
                }

                public List<ResolvedExpression> getRemainingFilters() {
                        return remainingFilters;
                }
        }
}

/**
 * Allows to push down (possibly nested) projections into a {@link ScanTableSource}.
 */
@PublicEvolving
public interface SupportsProjectionPushDown {

        boolean supportsNestedProjectionPushedDown();

        void applyProjection(TableSchema schema);
}
```

LookupTableSource

`LookupTableSource` supports a similar abstraction of planning and runtime code via `LookupRuntimeProvider`.

```
/**
 * A {@link DynamicTableSource} that looks up rows of an external storage system by one or more
 * keys.
 */
@PublicEvolving
public interface LookupTableSource extends DynamicTableSource {

        /**
         * Returns the actual implementation for reading the data.
         */
        LookupRuntimeProvider getLookupRuntimeProvider(Context context);

        // --------------------------------------------------------------------------------------------
        // Helper Interfaces
        // --------------------------------------------------------------------------------------------

        interface Context {

                /**
                 * Returns the key fields that should be used during the lookup.
                 */
                List<FieldReferenceExpression> getKeyFields();

                /**
                 * Creates a runtime data structure converter that converts data of the given {@link DataType}
                 * to Flink's internal data structures.
                 *
                 * <p>Allows to implement runtime logic without depending on Flink's internal structures for
                 * timestamps, decimals, and structured types.
                 *
                 * @see LogicalType#supportsInputConversion(Class)
                 */
                DataStructureConverter createDataStructureConverter(DataType producedDataType);
        }

        interface LookupRuntimeProvider {
                // marker interface
        }
}

/**
 * Uses a {@link TableFunction} during runtime for reading.
 */
public interface TableFunctionProvider<T> extends LookupTableSource.LookupRuntimeProvider {

        TableFunction<T> createTableFunction();

        static <T> TableFunctionProvider<T> of(TableFunction<T> tableFunction) {
                return () -> tableFunction;
        }
}

/**
 * Uses a {@link AsyncTableFunction} during runtime for reading.
 */
public interface AsyncTableFunctionProvider<T> extends LookupTableSource.LookupRuntimeProvider {

        AsyncTableFunction<T> createAsyncTableFunction();

        static <T> AsyncTableFunctionProvider<T> of(AsyncTableFunction<T> tableFunction) {
                return () -> tableFunction;
        }
}
```

Sink Interfaces

`DynamicTableSink` works similar to `ScanTableSource`. A more complex interface hierarchy is not required currently.

A big difference between the new `DynamicTableSink` and old `UpsertStreamTableSink` is that `setKeyFields` is not present anymore. This means that key information is not derived from each query individually but is statically defined by `CatalogTable#getSchema` using a primary key.

```
/**
 * Sink of a dynamic table to an external storage system.
 *
 * <p>A dynamic table sink can be seen as a factory that produces concrete runtime implementation.
 *
 * <p>Depending on optionally declared interfaces such as {@link SupportsPartitioning}, the planner
 * might apply changes to instances of this class and thus mutates the produced runtime
 * implementation.
 */
@PublicEvolving
public interface DynamicTableSink {

        /**
         * Returns a string that summarizes this sink for printing to a console or log.
         */
        String asSummaryString();

        /**
         * Returns the {@link ChangelogMode} that this writer consumes.
         *
         * <p>The runtime can make suggestions but the sink has the final decision what it requires. If
         * the runtime does not support this mode, it will throw an error. For example, the sink can
         * return that it only supports {@link RowKind#INSERT}s.
         *
         * @param requestedMode expected kind of changes by the current plan
         */
        ChangelogMode getChangelogMode(ChangelogMode requestedMode);

        /**
         * Returns the actual implementation for writing the data.
         */
        SinkRuntimeProvider getSinkRuntimeProvider(Context context);

        // --------------------------------------------------------------------------------------------
        // Helper Interfaces
        // --------------------------------------------------------------------------------------------

        interface Context {

                /**
                 * Creates a runtime data structure converter that converts Flink's internal data structures
                 * to data of the given {@link DataType}.
                 *
                 * <p>Allows to implement runtime logic without depending on Flink's internal structures for
                 * timestamps, decimals, and structured types.
                 *
                 * @see LogicalType#supportsOutputConversion(Class)
                 */
                DataStructureConverter createDataStructureConverter(DataType consumedDataType);
        }

        /**
         * Converts data structures during runtime.
         */
        interface DataStructureConverter extends RuntimeConverter {

                /**
                 * Converts the given object into an external data structure.
                 */
                @Nullable Object toExternal(@Nullable Object internalStructure);
        }

        interface SinkRuntimeProvider {
                // marker interface
        }
}
```

Factory Interfaces

Because we need new factories for the new source and sink interfaces, it is the right time to look at the big picture of how factories, connectors, and their formats play together.

Also, with the recent decision of unifying the Flink configuration experience by only using `ConfigOption`, the old factory interfaces with the `DescriptorProperties` util needs an update.

Factory

Because `TableFactory` was the superclass of all `flink-table` related factories, it confused users to use a table factory for modules, catalogs (not table related). The `supportedProperties` caused a long list of keys also for schema and format wildcards, thus, caused a lot of duplicate code.

The validation was partially done by the factory service and the factory itself. We suggest to perform the validation only in the factory. By using `CatalogTable` directly, the schema part must not be validated anymore.

For simplifying the discovery, we only match by one identifier (+ version). In theory, we could also use the class name itself, but this would make it more difficult to insert a custom factory without changing DDL properties.

```
/**
 * A factory for creating instances from {@link ConfigOption}s in the table ecosystem. This
 * factory is used with Java's Service Provider Interfaces (SPI) for discovery.
 */
@PublicEvolving
public interface Factory {

        /**
         * Uniquely identifies this factory when searching for a matching factory. Possibly versioned
         * by {@link #factoryVersion()}.
         */
        String factoryIdentifier();

        /**
         * Extends a {@link #factoryIdentifier()} by a version when searching for a matching factory.
         */
        Optional<String> factoryVersion();

        /**
         * Definition of required options for this factory. The information will be used for generation
         * of documentation and validation. It does not influence the discovery of a factory.
         */
        Set<ConfigOption<?>> requiredOptions();

        /**
         * Definition of optional options for this factory. This information will be used for generation
         * of documentation and validation. It does not influence the discovery of a factory.
         */
        Set<ConfigOption<?>> optionalOptions();
}
```

DynamicTableFactory, DynamicTableSourceFactory, DynamicTableSinkFactory

```
/**
 * Factory for accessing a dynamic table for reading or writing.
 */
public interface DynamicTableFactory extends Factory {

        /**
         * Information about the accessed table.
         */
        interface Context {

                /**
                 * Identifier of the table in the catalog.
                 */
                ObjectIdentifier getObjectIdentifier();

                /**
                 * Table information received from the {@link Catalog}.
                 */
                CatalogTable getCatalogTable();

                /**
                 * Configuration of the current session.
                 */
                ReadableConfig getConfiguration();

                /**
                 * Class loader of the current session.
                 */
                ClassLoader getClassLoader();
        }
}

/**
 * Factory for reading a dynamic table.
 */
public interface DynamicTableSourceFactory extends DynamicTableFactory {

        /**
         * Factory method for creating a {@link DynamicTableSource}.
         */
        DynamicTableSource createDynamicTableSource(Context context);
}

/**
 * Factory for writing a dynamic table.
 */
public interface DynamicTableSinkFactory extends DynamicTableFactory {

        /**
         * Factory method for creating a {@link DynamicTableSink}.
         */
        DynamicTableSink createDynamicTableSource(Context context);
}
```

FormatFactory, DeserializationFormatFactory

For a smooth interplay, we also need to consider the relationship between connectors and formats.

We propose the following format interfaces. And illustrate their usage in the following example.

```
/**
 * Base interface for all kinds of formats.
 */
public interface FormatFactory extends Factory {

        /**
         * Determines the changelog mode for this format.
         */
        ChangelogMode deriveChangelogMode(DynamicTableFactory.Context tableContext);
}

/**
 * Factory for creating a {@link DeserializationSchema} that returns internal data structures.
 */
public interface DeserializationFormatFactory extends FormatFactory {

        /**
         * Creates {@link DeserializationSchema} for the given produced {@link DataType} considering all
         * contextual information.
         */
        DeserializationSchema<RowData> createDeserializationSchema(
                DynamicTableFactory.Context tableContext,
                ScanTableSource.Context runtimeContext,
                DataType producedDataType);
}
```

For MySQL or Postgres CDC logs, the format should be able to return a `ChangelogMode`. Formats that don't produce changes, can simply return an insert-only changelog mode.

Because the final schema that formats need to handle is only known after optimization, we postpone instantiating concrete runtime interfaces such as `DeserializationSchema` to the `getXXXRuntimeProvider` methods.

However, the discovery and validation of the factory is done earlier.

```
class KafkaSourceFactory extends DynamicTableSourceFactory {

        // ...

        DynamicTableSource createDynamicTableSource(Context context) {
                // perform format factory discovery
                TableFormatFactory keyFormatFactory = FactoryUtil.find(TableFormatFactory.class, context,
KEY_OPTION);
                TableFormatFactory valueFormatFactory = FactoryUtil.find(TableFormatFactory.class, context,
VALUE_OPTION);

                // validate using required and optional options of each factory
                // also validates if there are left-over keys
                FactoryUtil.validate(
                        context,
                        Arrays.asList(KEY_OPTION, VALUE_OPTION),
                        Arrays.asList(this, keyFormatFactory, valueFormatFactory));

                ChangelogMode mode = valueFormatFactory.createChangelogMode(context);

                // construct the source
                return KafkaTableSource.builder(context)
                        .changelogMode(mode) // or extract it internally from the valueFormatFactory
                        .keyFormatFactory(keyFormatFactory)
                        .valueFormatFactory(valueFormatFactory)
                        .build();
        }
}

class FactoryUtil {
        // ...

        void validate(
                DynamicTableFactory.Context context,
                List<ConfigOption> usedOptions,
                List<Factory> usedFactories) {

                // perform ConfigOption set arithmetic to check for
                // invalid, missing options etc.
        }
}
```

Data Structure Interfaces

We plan to move the data structures of the Blink planner to `table-common`. In order to provide stable interfaces for data structures, we renamed the interfaces and reduced the exposed methods to have them in a good shape and mark them as `@PublicEvolving`.

All data structure names will be suffixed with `Data` for avoiding naming clashes with JDK interfaces, and indicating they are a series of structures to represent “data” for different types. Some well-known interfaces/classes are rename to:

- BaseRow -> RowData
- BaseArray -> ArrayData
- BaseMap -> MapData
- BinaryString -> StringData
- Decimal -> DecimalData
- SqlTimestamp -> TimestampData
- BinaryGeneric -> RawValueData
- GenericRow -> GenericRowData
- GenericArray -> GenericArrayData
- GenericMap -> GenericMapData

Note that all the internal data structures are not Java Serializable, so they can’t be used as member fields of runtime functions (e.g. SourceFunction).

We will not introduce a data structure interface/class for every data type, but will use Java primitive types as much as possible for performance purposes. The mappings from Flink Table/SQL data types to the internal data structures are listed in the following table:

Data Types	**Internal Data Structures**
BOOLEAN	boolean
CHAR / VARCHAR / STRING	StringData
BINARY / VARBINARY / BYTES	byte[]
DECIMAL	DecimalData
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
FLOAT	float
DOUBLE	double
DATE	int (number of days since epoch)
TIME	int (number of milliseconds of the day)
TIMESTAMP	TimestampData
TIMESTAMP WITH LOCAL TIME ZONE	TimestampData
INTERVAL YEAR TO MONTH	int (number of months)
INTERVAL DAY TO MONTH	long (number of milliseconds)
ROW	RowData
ARRAY	ArrayData
MAP / MULTISET	MapData
RAW	RawValueData

Note: Currently, Blink planner only supports TIME(0) and interval of MONTH and SECOND(3). In the future, we may need something like `TimestampData` to represent more information about these types. That means we may introduce `IntervalData` and `TimeData` in the future. But we still can keep backward compatibility to allow users to use the primitive types if the precision is compact at that time.

RowData

```
/**
 * {@link RowData} is an internal data structure representing data of {@link RowType}
 * in Flink Table/SQL, which only contains columns of the internal data structures.
 *
 * <p>A {@link RowData} also contains a {@link RowKind} which represents the kind of row in
 * a changelog. The {@link RowKind} is just metadata information of the row, not a column.
 *
 * <p>{@link RowData} has different implementations which are designed for different scenarios.
 * For example, the binary-oriented implementation {@link BinaryRowData} is backed by
 * {@link MemorySegment} instead of Object to reduce serialization/deserialization cost.
 * The object-oriented implementation {@link GenericRowData} is backed by an array of Object
 * which is easy to construct and efficient to update.
 */
@PublicEvolving
public interface RowData {

    int getArity();

    RowKind getRowKind();

    void setRowKind(RowKind kind);

    // ------------------------------------------------------------

    boolean isNullAt(int ordinal);
    boolean getBoolean(int ordinal);
    byte getByte(int ordinal);
    short getShort(int ordinal);
    int getInt(int ordinal);
    long getLong(int ordinal);
    float getFloat(int ordinal);
    double getDouble(int ordinal);
    StringData getString(int ordinal);
    DecimalData getDecimal(int ordinal, int precision, int scale);
    TimestampData getTimestamp(int ordinal, int precision);
    <T> RawValueData<T> getRawValue(int ordinal);
    byte[] getBinary(int ordinal);
    ArrayData getArray(int ordinal);
    MapData getMap(int ordinal);
    RowData getRow(int ordinal, int numFields);
}
```

ArrayData

```
/**
 * {@link ArrayData} is an internal data structure representing data of {@link ArrayType}
 * in Flink Table/SQL, which only contains elements of the internal data structures.
 */
@PublicEvolving
public interface ArrayData {

    int size();

    // ------------------------------------------------------------------------

    boolean isNullAt(int ordinal);
    boolean getBoolean(int ordinal);
    byte getByte(int ordinal);
    short getShort(int ordinal);
    int getInt(int ordinal);
    long getLong(int ordinal);
    float getFloat(int ordinal);
    double getDouble(int ordinal);
    StringData getString(int ordinal);
    DecimalData getDecimal(int ordinal, int precision, int scale);
    TimestampData getTimestamp(int ordinal, int precision);
    <T> RawValueData<T> getGeneric(int ordinal);
    byte[] getBinary(int ordinal);
    ArrayData getArray(int ordinal);
    MapData getMap(int ordinal);
    RowData getRow(int ordinal, int numFields);
}
```

MapData

```
/**
 * {@link MapData} is an internal data structure representing data of {@link MapType}
 * in Flink Table/SQL.
 */
@PublicEvolving
public interface MapData {

    int size();

    ArrayData keyArray();

    ArrayData valueArray();
}
```

StringData

```
/**
 * {@link StringData} is an internal data structure represents data of {@link VarCharType}
 * and {@link CharType} in Flink Table/SQL.
 */
@PublicEvolving
public interface StringData extends Comparable<StringData> {

    /**
     * Converts this {@link StringData} object to a UTF-8 byte array,
     * the returned bytes may be reused.
     */
    byte[] toBytes();

    /**
     * Converts this {@link StringData} object to a {@link String} and returns the String.
     */
    String toString();

    // ------------------------------------------------------------------------------------------

    /**
     * Creates a {@link StringData} from the given String.
     */
    static StringData fromString(String str) { ... }

    /**
     * Creates a {@link StringData} from the given UTF-8 bytes.
     */
    static StringData fromBytes(byte[] bytes) { ... }

    /**
     * Creates a {@link StringData} from the given UTF-8 bytes with offset and number of bytes.
     */
    static StringData fromBytes(byte[] bytes, int offset, int numBytes) { ... }
}
```

DecimalData

```
/**
 * {@link DecimalData} is an internal data structure representing data of {@link DecimalType}
 * in Flink Table/SQL.
 *
 * <p>It is an immutable implementation which can hold a long if values are small enough.
 */
@PublicEvolving
public final class DecimalData implements Comparable<DecimalData> {

    public int precision() { ... }
    public int scale() { ... }
    public BigDecimal toBigDecimal() { ... }

    /**
     * Returns a long whose value is the <i>unscaled value</i> of this {@code DecimalData}.
     */
    public long toUnscaledLong() { ... }

    /**
     * Returns a byte array whose value is the <i>unscaled value</i> of
     * this {@code DecimalData}.
     */
    public byte[] toUnscaledBytes() { ... }

    /**
     * Returns whether the decimal data is small enough to be stored in a long.
     */
    public boolean isCompact() { ... }

    public DecimalData copy() { ... }

    // ------------------------------------------------------------------------

    public static DecimalData fromBigDecimal(BigDecimal bd, int precision, int scale) {
      ...
    }

    public static DecimalData fromUnscaledLong(int precision, int scale, long longVal) {
      ...
    }

    public static DecimalData fromUnscaledBytes(int precision, int scale, byte[] bytes) {
      ...
    }

    public static DecimalData zero(int precision, int scale) { ... }
}
```

TimestampData

```
/**
 * {@link TimestampData} is an internal data structure represents data of {@link TimestampType}
 * and {@link LocalZonedTimestampType} in Flink Table/SQL.
 *
 * <p>It is an immutable implementation which is composite of a millisecond
 * and nanoOfMillisecond since epoch.
 */
@PublicEvolving
public final class TimestampData implements Comparable<TimestampData> {

    public long getMillisecond() { ... }
    public int getNanoOfMillisecond() { ... }
    public Timestamp toTimestamp() { ... }
    public LocalDateTime toLocalDateTime() { ... }
    public Instant toInstant() { ... }

    // ------------------------------------------------------------------------

    public static TimestampData fromEpochMillis(long millisecond) { ... }

    public static TimestampData fromEpochMillis(long millisecond, int nanoOfMillisecond) {
      ...
    }

    public static TimestampData fromTimestamp(Timestamp ts) { ... }

    public static TimestampData fromLocalDateTime(LocalDateTime dateTime) { ... }

    public static TimestampData fromInstant(Instant instant) { ... }

    /**
     * Returns whether the timestamp data is small enough to be stored in a long
     * of millisecond.
     */
    public static boolean isCompact(int precision) { ... }
}
```

RawValueData

```
/**
 * {@link RawValueData} is a data structure representing data of {@link RawType}
 * in Flink Table/SQL.
 *
 * @param <T> originating class for the raw value
 */
@PublicEvolving
public interface RawValueData<T> {

    /**
     * Converts a {@link RawValueData} into a Java object, the {@code serializer}
     * is required because the "raw value" might be in binary format which can be
     * deserialized by the {@code serializer}.
     *
     * Note: the returned Java object may be reused.
     */
    T toObject(TypeSerializer<T> serializer);

    /**
     * Converts a {@link RawValueData} into a byte array, the {@code serializer}
     * is required because the "raw value" might be in Java object format which
     * can be serialized by the {@code serializer}.
     *
     * Note: the returned bytes may be reused.
     */
    byte[] toBytes(TypeSerializer<T> serializer);

    // ------------------------------------------------------------------------

    /**
     * Creates a {@link RawValueData} instance from a java object.
     */
    static <T> RawValueData<T> fromObject(T javaObject) { ... }

}
```

GenericRowData

```
/**
 * A {@link GenericRowData} can have arbitrary number of fields and contain a
 * set of fields, which may all be different types. The fields in
 * {@link GenericRowData} can be null.
 *
 * <p>The fields in the row can be accessed by position (zero-based) {@link #getInt}.
 * And can update fields by {@link #setField(int, Object)}.
 */
@PublicEvolving
public final class GenericRowData implements RowData {

    public GenericRowData(int arity) { ... }

    /**
     * Sets the field at the specified ordinal. The given field value must in
     * internal data structure, otherwise the {@link GenericRowData} is corrupted,
     * and may throw an exception when processing.
     */
    public void setField(int ordinal, Object value) { ... }


    /**
     * Gets the field at the specified ordinal. The returned field value is in
     * internal data structure.
     */
    public Object getField(int ordinal) { ... }

    ......

    // ----------------------------------------------------------

    /**
     * Creates a GenericRowData with the given internal data structure values and a default
     * {@link RowKind#INSERT}.
     *
     * @param values internal format values
     */
    public static GenericRowData of(Object... values) { ... }
}
```

GenericArrayData

```
@PublicEvolving
public final class GenericArrayData implements ArrayData {

    public GenericArrayData(Object[] array) { ... }

    public GenericArrayData(int[] primitiveArray) { ... }

    public GenericArrayData(long[] primitiveArray) { ... }

    public GenericArrayData(float[] primitiveArray) { ... }

    public GenericArrayData(double[] primitiveArray) { ... }

    public GenericArrayData(short[] primitiveArray) { ... }

    public GenericArrayData(byte[] primitiveArray) { ... }

    public GenericArrayData(boolean[] primitiveArray) { ... }

    public boolean isPrimitiveArray() { ... }

    ...

}
```

GenericMapData

```
@PublicEvolving
public final class GenericMapData implements MapData {

    public GenericMapData(Map<?, ?> map) { ... }

    ...
}
```

Proposed Changes

1. Move data structures of the Blink planner to `table-common` under `org.apache.flink.table.datastructures` package.

2. Introduce the new connector interfaces in `org.apache.flink.table.connectors.[sources/sinks]`, a parallel stack without affecting existing interfaces.
3. Update the first connectors such as Kafka, Hive and HBase to prove the new interfaces.
4. Enable `Row` to contain a change flag.

Compatibility, Deprecation, and Migration Plan

Because we introduce an entirely new stack of interfaces. Compatibility is not affected immediately.

Locations where `TableSource`s are exposed such as `fromTableSource()` are already deprecated or will be deprecated. It is recommended that users use the `connect()` API or DDL.

The legacy planner has different internal data structures and limited functionality in general. Therefore, we will not support the new interfaces in the legacy planner.

Test Plan

Unit tests will test the interfaces. Existing connector tests will verify the implementation.

Rejected Alternatives

Not moving Blink's data structures to `table-common`

We were considering using only converters or producers classes from `Context` for creating rows and other data structures, thus, hiding the internal data structures in `table-runtime-blink`.

However, we decided against this as most connectors are implemented in the Flink code base and will use binary formats for efficiency. The `table-common` module was meant as the only dependency that connectors need to implement in the future according to FLIP-32. Therefore, we will need to move those internal data structures.

The converters provided via the `Context`s provide an easy way of avoiding internal data structures and reuse legacy code from formats or connectors that were returning `Row`.

Discussions around filter pushdown/projection pushdown

https://docs.google.com/document/d/1vSwjx6LJsBFJUThEaTS00z3Nt5r5fBZ0id4SQbfBPyI/edit

Other discussions

https://docs.google.com/document/d/1ZizzC51L_f52e3mQkxbV5cfNq2K9w8OGq2AlMzKmv78/edit#heading=h.fdwd0zdiycb7