FLIP-175: Compose Estimator/Model/AlgoOperator from DAG of Estimator/Model/AlgoOperator (Flink ML)

**Discussion thread**  
https://lists.apache.org/thread/rfb3tglxb9f6jx8j6d6xb5scz5krxij4d

**Vote thread**  
https://lists.apache.org/thread/k6zo13tp59q16ogsg9xmjx5j9yp99dj9

**JIRA**  
[FLINK-23959](https://issues.apache.org/jira/browse/FLINK-23959) - FLIP-175: Compose Estimator/Model/AlgoOperator from DAG of Estimator/Model/AlgoOperator  
CLOSED

**Release**  
ml-2.0.0

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

- Motivation and Use-cases
- Public Interfaces
- Example Usage
  - Composing an Estimator from a DAG of Estimator/Transformer
  - Compose an Estimator from a chain of Estimator/Transformer whose input schemas are different from its fitted Transformer
- Compatibility, Deprecation, and Migration Plan
- Test Plan
- Rejected Alternatives

[This FLIP proposal is a joint work between Dong Lin and Zhipeng Zhang]

**Motivation and Use-cases**

The existing Flink ML library allows users to compose an Estimator/Transformer/AlgoOperator from a pipeline (i.e. linear sequence) of Estimator/Transformer/AlgoOperator. Users only need to construct this Pipeline once and generate the corresponding PipelineModel, without having to explicitly construct the fitted PipelineModel as a linear sequence of stages. However, in order to train a DAG of Estimator/Transformer/AlgoOperator and use the trained model for inference, users currently need to construct the DAG twice, once for the training logic and once for the inference logic. This experience is inferior to the experience of training and using a chain of Estimator/Transformer/AlgoOperator. In addition to requiring more work from users, this approach is more error prone because the DAG for the training logic may be inconsistent from the DAG for the inference logic.

In order to address the issues described above, we propose to add several helper classes that allow users to compose Estimator/Transformer/AlgoOperator from a DAG of Estimator/Transformer/AlgoOperator.

**Public Interfaces**

This FLIP proposes to add the Graph, GraphModel, GraphBuilder, GraphNode and TableId classes. The following code block shows the public APIs of these classes.

1) Add the TableId class to represent the input/output of a stage.

This class is necessary in order to construct the DAG before we have the concrete Tables available. And this class overrides the equals/hashCode so that it can be used as the key of a hash map.

```java
public class TableId {
    private final int tableId;

    @Override
    public boolean equals(Object obj) {...}

    @Override
    public int hashCode() {...}
}
```

2) Add the GraphNode class.

This class contains the stage as well as the input/output of this stage in the form of TableId lists. A DAG can thus be represented as a list of GraphNodes.
public class GraphNode {
    public final Stage<?> stage;
    public final TableId[] estimatorInputs;
    public final TableId[] algoOpInputs;
    public final TableId[] outputs;
}

3) Add the Graph class to wrap a DAG of Estimator/Model/Transformer/AlgoOperator into an Estimator.

/**
 * A Graph acts as an Estimator. A Graph consists of a DAG of stages, each of which could be an
 * Estimator, Model, Transformer or AlgoOperator. When `Graph::fit` is called, the stages are
 * executed in a topologically-sorted order. If a stage is an Estimator, its `Estimator::fit` method
 * will be called on the input tables (from the input edges) to fit a Model. Then the Model will be
 * used to transform the input tables and produce output tables to the output edges. If a stage is
 * an AlgoOperator, its `AlgoOperator::transform` method will be called on the input tables and
 * produce output tables to the output edges. The GraphModel fitted from a Graph consists of the
 * fitted Models and AlgoOperators, corresponding to the Graph's stages.
 */
@PublicEvolving
public final class Graph implements Estimator<Graph, GraphModel> {
    public Graph(List<GraphNode> nodes, TableId[] estimatorInputIds, TableId[] algoOpInputs, TableId[] outputs,
                 TableId[] inputModelData, TableId[] outputModelData) {...}

    @Override
    public GraphModel fit(Table... inputs) {...}

    @Override
    public void save(String path) throws IOException {...}

    @Override
    public static Graph load(StreamTableEnvironment tEnv, String path) throws IOException {...}
}

4) Add the GraphModel class to wrap a DAG of Estimator/Model/Transformer/AlgoOperator into a Model.

/**
 * A GraphModel acts as a Model. A GraphModel consists of a DAG of stages, each of which could be an
 * Estimator, Model, Transformer or AlgoOperators. When `GraphModel::transform` is called, the
 * stages are executed in a topologically-sorted order. When a stage is executed, its
 * `AlgoOperator::transform` method will be called on the input tables (from the input edges) and
 * produce output tables to the output edges.
 */
public final class GraphModel implements Model<GraphModel> {
    public GraphModel(List<GraphNode> nodes, TableId[] inputIds, TableId[] outputIds, TableId[] inputModelData,
                      TableId[] outputModelData) {...}

    @Override
    public Table[] transform(Table... inputTables) {...}

    @Override
    public void setModelData(Table... inputs) {...}

    @Override
    public Table[] getModelData() {...}

    @Override
    public void save(String path) throws IOException {...}

    public static GraphModel load(StreamTableEnvironment tEnv, String path) throws IOException {...}
5) Add the GraphBuilder class to build GraphModel or Graph from a DAG of stages.

```java
/**
 * A GraphBuilder provides APIs to build Estimator/Model/AlgoOperator from a DAG of stages, each of
 * which could be an Estimator, Model, Transformer or AlgoOperator.
 */
@PublicEvolving
public final class GraphBuilder {

    /**
     * Specifies the loose upper bound of the number of output tables that can be returned by the
     * Model::getModelData() and AlgoOperator::transform() methods, for any stage involved in this
     * Graph.
     *
     * <p>The default upper bound is 20.
     */
    public GraphBuilder setMaxOutputTableNum(int maxOutputLength) {...}

    /**
     * Creates a TableId associated with this GraphBuilder. It can be used to specify the passing of
     * tables between stages, as well as the input/output tables of the Graph/GraphModel generated
     * by this builder.
     *
     * @return A TableId.
     */
    public TableId createTableId() {...}

    /**
     * Adds an AlgoOperator in the graph.
     *
     * <p>When the graph runs as Estimator, the transform() of the given AlgoOperator would be
     * invoked with the given inputs. Then when the GraphModel fitted by this graph runs, the
     * transform() of the given AlgoOperator would be invoked with the given inputs.
     *
     * <p>When the graph runs as AlgoOperator or Model, the transform() of the given AlgoOperator
     * would be invoked with the given inputs.
     *
     * <p>NOTE: the number of the returned TableIds does not represent the actual number of Tables
     * outputted by transform(). This number could be configured using {@link
     * #setMaxOutputTableNum(int)}. Users should make sure that this number >= the actual number of
     * Tables outputted by transform().
     *
     * @param algoOp An AlgoOperator instance.
     * @param inputs A list of TableIds which represents inputs to transform() of the given
     * AlgoOperator.
     * @return A list of TableIds which represents the outputs of transform() of the given
     * AlgoOperator.
     */
    public TableId[] addAlgoOperator(AlgoOperator<?> algoOp, TableId... inputs) {...}

    /**
     * Adds an Estimator in the graph.
     *
     * <p>When the graph runs as Estimator, the fit() of the given Estimator would be invoked with
     * the given inputs. Then when the GraphModel fitted by this graph runs, the transform() of the
     * Model fitted by the given Estimator would be invoked with the given inputs.
     *
     * <p>When the graph runs as AlgoOperator or Model, the fit() of the given Estimator would be
     * invoked with the given inputs, then the transform() of the Model fitted by the given
     * Estimator would be invoked with the given inputs.
     *
     * <p>NOTE: the number of the returned TableIds does not represent the actual number of Tables
     * outputted by transform(). This number could be configured using (@link
     * #setMaxOutputTableNum(int)). Users should make sure that this number >= the actual number of
     * Tables outputted by transform().
     *
     * @param estimator An Estimator instance.
     * @param inputs A list of TableIds which represents inputs to fit() of the given Estimator as
     */
    public void addEstimator(Estimator<?> estimator, TableId... inputs) {...}
}
```
public TableId[] addEstimator(Estimator<?, ?> estimator, TableId... inputs) {...}

/**
 * Adds an Estimator in the graph.
 *<p>
 *When the graph runs as Estimator, the fit() of the given Estimator would be invoked with estimatorInputs. Then when the GraphModel fitted by this graph runs, the transform() of the Model fitted by the given Estimator would be invoked with modelInputs.
 *<p>
 *When the graph runs as AlgoOperator or Model, the fit() of the given Estimator would be invoked with estimatorInputs, then the transform() of the Model fitted by the given Estimator would be invoked with modelInputs.
 *<p>
 *NOTE: the number of the returned TableIds does not represent the actual number of Tables outputted by transform(). This number could be configured using {@link #setMaxOutputTableNum(int)}. Users should make sure that this number >= the actual number of Tables outputted by transform().
 */
public TableId[] addEstimator(Estimator<?, ?> estimator, TableId[] estimatorInputs, TableId[] modelInputs) {...}

/**
 * When the graph runs as Estimator, it first generates a GraphModel that contains the Model fitted by the given Estimator. Then when this GraphModel runs, the setModelData() of the fitted Model would be invoked with the given inputs before its transform() is invoked.
 *<p>
 *When the graph runs as AlgoOperator or Model, the setModelData() of the Model fitted by the given Estimator would be invoked with the given inputs before its transform() is invoked.
 */
public void setModelDataOnEstimator(Estimator<?, ?> estimator, TableId... inputs) {...}

/**
 * When the graph runs as Estimator, the setModelData() of the given Model would be invoked with the given inputs before its transform() is invoked. Then when the GraphModel fitted by this graph runs, the setModelData() of the fitted Model would be invoked with the given inputs.
 *<p>
 *When the graph runs as AlgoOperator or Model, the setModelData() of the given Model would be invoked with the given inputs before its transform() is invoked.
 *<p>
 *NOTE: the number of the returned TableIds does not represent the actual number of Tables outputted by getModelProperty(). This number could be configured using {@link
*/
public void setModelDataOnModel(Model<?> model, TableId... inputs) {...}
* @param estimator An Estimator instance.
* @return A list of TableIds which represents the outputs of getModelData() of the Model fitted
* by the given Estimator.
*/
public TableId[] getModelDataFromEstimator(Estimator<?, ?> estimator) {...}

/**
* When the graph runs as Estimator, the getModelData() of the given Model would be invoked.
* Then when the GraphModel fitted by this graph runs, the getModelData() of the given Model
* would be invoked.
*<p>When the graph runs as AlgoOperator or Model, the getModelData() of the given Model
* would be invoked.
*<p>NOTE: the number of the returned TableIds does not represent the actual number of Tables
* outputted by getModelData(). This number could be configured using {#setMaxOutputTableNum(int)}. Users should make sure that this number >= the actual number of
* Tables outputted by getModelData().
* @param model A Model instance.
* @return A list of TableIds which represents the outputs of getModelData() of the given Model.
*/
public TableId[] getModelDataFromModel(Model<?> model) {...}

/**
* Wraps nodes of the graph into an Estimator.
*<p>When the returned Estimator runs, and when the Model fitted by the returned Estimator
* runs, the sequence of operations recorded by the {#addAlgoOperator(...)}, {#addEstimator(...)},
* {#setModelData(...)} and {#getModelData(...)} would be executed as specified in the Java doc of the corresponding methods.
* @param inputs A list of TableIds which represents inputs to fit() of the returned Estimator
* as well as inputs to transform() of the Model fitted by the returned Estimator.
* @param outputs A list of TableIds which represents outputs of transform() of the Model fitted
* by the returned Estimator.
* @return An Estimator which wraps the nodes of this graph.
*/
public Estimator<?, ?> buildEstimator(TableId[] inputs, TableId[] outputs) {...}

/**
* Wraps nodes of the graph into an Estimator.
*<p>When the returned Estimator runs, and when the Model fitted by the returned Estimator
* runs, the sequence of operations recorded by the {#addAlgoOperator(...)}, {#addEstimator(...)},
* {#setModelData(...)} and {#getModelData(...)} would be executed as specified in the Java doc of the corresponding methods.
* @param inputs A list of TableIds which represents inputs to fit() of the returned Estimator
* as well as inputs to transform() of the Model fitted by the returned Estimator.
* @param outputs A list of TableIds which represents outputs of transform() of the Model fitted
* by the returned Estimator.
* @param inputModelData A list of TableIds which represents inputs to setModelData() of the
* Model fitted by the returned Estimator.
* @param outputModelData A list of TableIds which represents outputs of getModelData() of the
* Model fitted by the returned Estimator.
* @return An Estimator which wraps the nodes of this graph.
*/
public Estimator<?, ?> buildEstimator(TableId[] inputs, TableId[] outputs, TableId[] inputModelData, TableId[] outputModelData) {...}
* runs, the sequence of operations recorded by the {code addAlgoOperator(...)}, {code
  addEstimator(...)}, {code setModelData(...)} and {code getModelData(...)} would be executed
* as specified in the Java doc of the corresponding methods.
*
* @param estimatorInputs A list of TableIds which represents inputs to fit() of the returned
  * Estimator.
* @param modelInputs A list of TableIds which represents inputs to transform() of the Model
  * fitted by the returned Estimator.
* @param outputs A list of TableIds which represents outputs of transform() of the Model fitted
  * by the returned Estimator.
* @param inputModelData A list of TableIds which represents inputs to setModelData() of the
  * Model fitted by the returned Estimator.
* @param outputModelData A list of TableIds which represents outputs of getModelData() of the
  * Model fitted by the returned Estimator.
* @return An Estimator which wraps the nodes of this graph.
*/
public Estimator<?, ?> buildEstimator(
    TableId[] estimatorInputs,
    TableId[] modelInputs,
    TableId[] outputs,
    TableId[] inputModelData,
    TableId[] outputModelData) {...}

/**
 * Wraps nodes of the graph into an AlgoOperator.
 *
 * <p>When the returned AlgoOperator runs, the sequence of operations recorded by the {code
 * addAlgoOperator(...)} and {code addEstimator(...)} would be executed as specified in the
 * Java doc of the corresponding methods.
 *
 * @param inputs A list of TableIds which represents inputs to transform() of the returned
  * AlgoOperator.
* @param outputs A list of TableIds which represents outputs of transform() of the returned
  * AlgoOperator.
* @return An AlgoOperator which wraps the nodes of this graph.
*/
public AlgoOperator<?> buildAlgoOperator(TableId[] inputs, TableId[] outputs) {...}

/**
 * Wraps nodes of the graph into a Model.
 *
 * <p>When the returned Model runs, the sequence of operations recorded by the {code
 * addAlgoOperator(...)}, {code addEstimator(...)}, {code setModelData(...)} and {code
 * getModelData(...)} would be executed as specified in the Java doc of the corresponding
 * methods.
 *
 * @param inputs A list of TableIds which represents inputs to transform() of the returned
  * Model.
* @param outputs A list of TableIds which represents outputs of transform() of the returned
  * Model.
* @param inputModelData A list of TableIds which represents inputs to setModelData() of the
  * returned Model.
* @param outputModelData A list of TableIds which represents outputs of getModelData() of the
  * returned Model.
* @return A Model which wraps the nodes of this graph.
*/
public Model<?, ?> buildModel(TableId[] inputs, TableId[] outputs) {...}
Example Usage

In this section we provide examples code snippets to demonstrate how we can use the APIs proposed in this FLIP to address the use-cases in the motivation section.

Composing an Estimator from a DAG of Estimator/Transformer

Suppose we have the following Transformer and Estimator classes:

- TransformerA whose transform(...) takes 1 input table and has 1 output table.
- ModelB whose transform(...) takes 2 input tables and has 1 output table.
- EstimatorB whose fit(...) takes 2 input tables and returns an instance of ModelB.

And we want to compose an Estimator (e.g. Graph) from the following DAG of Transformer/Estimator.

The resulting Graph::fit is expected to have the following behavior:

- The method takes 2 input tables. The 1st input table is given to a TransformerA instance. And the 2nd input table is given to another TransformerA instance.
- An EstimatorB instance fits the output tables of these two TransformerA instances and generates a new ModelB instance.
- Returns a GraphModel instance which contains 2 TransformerA instance and 1 ModelB instance, connected using the same DAG as shown above.

Here is the code snippet that addresses this use-case by using the proposed APIs:
GraphBuilder builder = new GraphBuilder();

// Creates nodes
AlgoOperator<?> stage1 = new TransformerA();
AlgoOperator<?> stage2 = new TransformerA();
Estimator<?> stage3 = new EstimatorB();

// Creates inputs and inputStates
TableId input1 = builder.createTableId();
TableId input2 = builder.createTableId();

// Feeds inputs to nodes and gets outputs.
TableId output1 = builder.addAlgoOperator(stage1, input1)[0];
TableId output2 = builder.addAlgoOperator(stage2, input2)[0];
TableId output3 = builder.addEstimator(stage3, output1, output2)[0];

// Specifies the ordered lists of inputs, outputs, input states and output states that will
// be used as the inputs/outputs of the corresponding Graph and GraphTransformer APIs.
TableId[] inputs = new TableId[] {input1, input2};
TableId[] outputs = new TableId[] {output3};

// Generates the Graph instance.
Estimator<?, ?> estimator = builder.buildEstimator(inputs, outputs);
// The fit method takes 2 tables which are mapped to input1 and input2.
Model<?, ?> model = estimator.fit(...);
// The transform method takes 2 tables which are mapped to input1 and input2.
Table[] results = model.transform(...);

Compose an Estimator from a chain of Estimator/Transformer whose input schemas are
different from its fitted Transformer

Suppose we have the following Estimator and Transformer classes where an Estimator's input schemas could be different from the input schema of its
fitted Transformer:

- TransformerA whose transform(...) takes 1 input table and has 1 output table.
- EstimatorA whose fit(...) takes 2 input tables and returns an instance of TransformerA.
- TransformerB whose transform(...) takes 1 input table and has 1 output table.

And we want to compose an Estimator (e.g. Graph) from the following DAG of Transformer/Estimator.

![DAG Diagram]

The resulting Graph::fit is expected to have the following behavior:

- The method takes 2 input tables. Both tables are given to EstimatorA::fit.
- EstimatorA fits the input tables and generates a TransformerA instance. The TransformerA instance takes 1 table input, which is different from the
  2 tables given to the EstimatorA.
- Returns a GraphModel instance which contains a TransformerA instance and a TransformerB instance, which are connected as a chain.

The fitted GraphModel is represented by the following DAG:

![DAG Diagram]

Here is the code snippet that addresses this use-case by using the proposed APIs:
GraphBuilder builder = new GraphBuilder();

// Creates nodes
Estimator<?> stage1 = new EstimatorA();
AlgoOperator<?> stage2 = new TransformerB();
// Creates inputs
TableId estimatorInput1 = builder.createTableId();
TableId estimatorInput2 = builder.createTableId();
TableId transformerInput1 = builder.createTableId();

// Feeds inputs to nodes and gets outputs.
TableId output1 = builder.addEstimator(stage1, new TableId[] {estimatorInput1, estimatorInput2}, new TableId[] {transformerInput1})[0];
TableId output2 = builder.addAlgoOperator(stage2, output1)[0];

// Specifies the ordered lists of estimator inputs, transformer inputs, outputs, input states and output states
// that will be used as the inputs/outputs of the corresponding Graph and GraphTransformer APIs.
TableId[] estimatorInputs = new TableId[] {estimatorInput1, estimatorInput2};
TableId[] transformerInputs = new TableId[] {transformerInput1};
TableId[] outputs = new TableId[] {output2};
TableId[] inputModelData = new TableId[] {};
TableId[] outputModelData = new TableId[] {};

// Generates the Graph instance.
Estimator<?, ?> estimator = builder.buildEstimator(estimatorInputs, transformerInputs, outputs, inputModelData, outputModelData);
// The fit method takes 2 tables which are mapped to estimatorInput1 and estimatorInput2.
Model<?> model = estimator.fit(...);
// The transform method takes 1 table which is mapped to transformerInput1.
Table[] results = model.transform(...);

Compatibility, Deprecation, and Migration Plan

This FLIP does not remove or modify any existing APIs. There is no backward incompatible change, deprecation or migration plan.

Test Plan

We will provide unit tests to validate the proposed changes.

Rejected Alternatives

There is no rejected alternatives to be listed here yet.