O'REILLY®

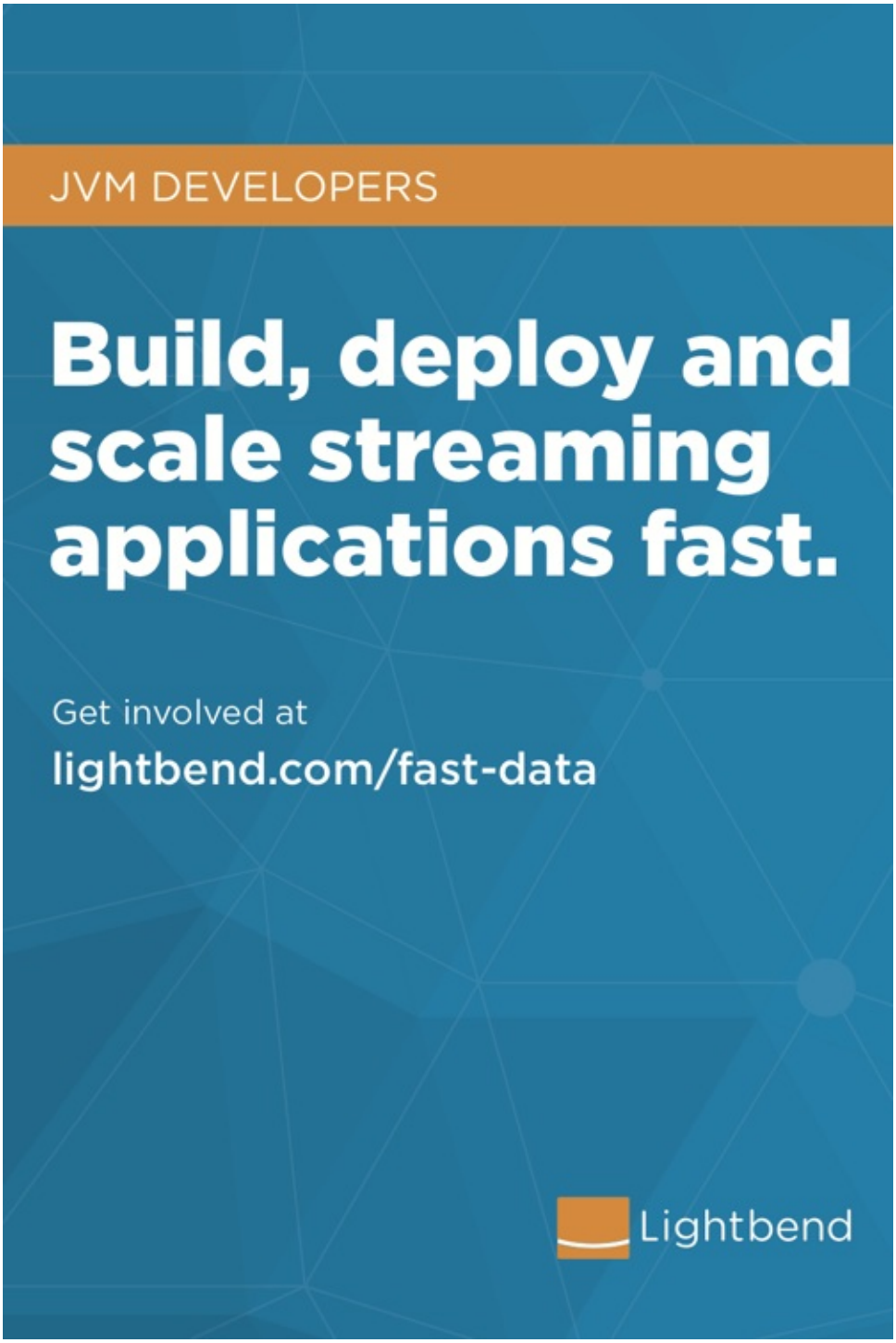



# Serving Machine Learning Models

## A Guide to Architecture, Stream Processing Engines, and Frameworks

Boris Lublinsky

# name of event



# Serving Machine Learning Models

A Guide to Architecture, Stream Processing Engines, and Frameworks

**Boris Lublinsky**


Beijing • Boston • Farnham • Sebastopol • Tokyo

## Serving Machine Learning Models

by Boris Lublinsky

Copyright © 2017 Lightbend, Inc. All rights reserved.

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com/safari). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Brian Foster & Virginia Wilson

Production Editor: Justin Billing

Copyeditor: Octal Publishing, Inc.

Proofreader: Charles Roumeliotis

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

October 2017: First Edition

## Revision History for the First Edition

- 2017-10-11: First Release

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Serving Machine Learning Models*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the

information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-02406-4

[LSI]

# Introduction

Machine learning is the hottest thing in software engineering today. There are a lot of publications on machine learning appearing daily, and new machine learning products are appearing all the time. Amazon, Microsoft, Google, IBM, and others have introduced machine learning as managed cloud offerings.

However, one of the areas of machine learning that is not getting enough attention is model serving—how to serve the models that have been trained using machine learning.

The complexity of this problem comes from the fact that typically model training and model serving are responsibilities of two different groups in the enterprise who have different functions, concerns, and tools. As a result, the transition between these two activities is often nontrivial. In addition, as new machine learning tools appear, it often forces developers to create new model serving frameworks compatible with the new tooling.

This book introduces a slightly different approach to model serving based on the introduction of standardized document-based intermediate representation of the trained machine learning models and using such representations for serving in a stream-processing context. It proposes an overall architecture implementing controlled streams of both data and models that enables not only the serving of models in real time, as part of processing of the input streams, but also enables updating models without restarting existing applications.

## Who This Book Is For

This book is intended for people who are interested in approaches to real-time serving of machine learning models supporting real-time model updates. It describes step-by-step options for exporting models, what exactly to export, and how to use these models for real-time serving.

The book also is intended for people who are trying to implement such solutions using modern stream processing engines and frameworks such as Apache Flink, Apache Spark streaming, Apache Beam, Apache Kafka streams, and Akka streams. It provides a set of working examples of usage of these technologies for model serving implementation.

## Why Is Model Serving Difficult?

When it comes to machine learning implementations, organizations typically employ two very different groups of people: data scientists, who are typically responsible for the creation and training models, and software engineers, who concentrate on model scoring. These two groups typically use completely different tools. Data scientists work with R, Python, notebooks, and so on, whereas software engineers typically use Java, Scala, Go, and so forth. Their activities are driven by different concerns: data scientists need to cope with the amount of data, data cleaning issues, model design and comparison, and so on; software engineers are concerned with production issues such as performance, maintainability, monitoring, scalability, and failover.

These differences are currently fairly well understood and result in many "proprietary" model scoring solutions, for example, Tensorflow model serving and Spark-based model serving. Additionally all of the managed machine learning implementations (Amazon, Microsoft, Google, IBM, etc.) provide model serving capabilities.

## Tools Proliferation Makes Things Worse

In his recent talk, Ted Dunning describes the fact that with multiple tools

available to data scientists, they tend to use different tools to solve different problems (because every tool has its own sweet spot and the number of tools grows daily), and, as a result, they are not very keen on tools standardization. This creates a problem for software engineers trying to use “proprietary” model serving tools supporting specific machine learning technologies. As data scientists evaluate and introduce new technologies for machine learning, software engineers are forced to introduce new software packages supporting model scoring for these additional technologies.

One of the approaches to deal with these problems is the introduction of an API gateway on top of the proprietary systems. Although this hides the disparity of the backend systems from the consumers behind the unified APIs, for model serving it still requires installation and maintenance of the actual model serving implementations.

## Model Standardization to the Rescue

To overcome these complexities, the Data Mining Group has introduced two model representation standards: Predictive Model Markup Language (PMML) and Portable Format for Analytics (PFA)

The Data Mining Group Defines PMML as:

> *is an XML-based language that provides a way for applications to define statistical and data-mining models as well as to share models between PMML-compliant applications.*
>
> *PMML provides applications a vendor-independent method of defining models so that proprietary issues and incompatibilities are no longer a barrier to the exchange of models between applications. It allows users to develop models within one vendor’s application, and use other vendors’ applications to visualize, analyze, evaluate or otherwise use the models. Previously, this was very difficult, but with PMML, the exchange of models between compliant applications is now straightforward. Because PMML is an XML-based standard, the specification comes in the form of an XML Schema.*

The Data Mining Group describes PFA as

> *an emerging standard for statistical models and data transformation engines.*

> *PFA combines the ease of portability across systems with algorithmic flexibility: models, pre-processing, and post processing are all functions that can be arbitrarily composed, chained, or built into complex workflows. PFA may be as simple as a raw data transformation or as sophisticated as a suite of concurrent data mining models, all described as a JSON or YAML configuration file.*

Another de facto standard in machine learning today is TensorFlow--an open-source software library for Machine Intelligence. Tensorflow can be defined as follows:

> *At a high level, TensorFlow is a Python library that allows users to express arbitrary computation as a graph of data flows. Nodes in this graph represent mathematical operations, whereas edges represent data that is communicated from one node to another. Data in TensorFlow are represented as tensors, which are multidimensional arrays.*

TensorFlow was released by Google in 2015 to make it easier for developers to design, build, and train deep learning models, and since then, it has become one of the most used software libraries for machine learning. You also can use TensorFlow as a backend for some of the other popular machine learning libraries, for example, Keras. TensorFlow allows for the exporting of trained models in protocol buffer formats (both text and binary) that you can use for transferring models between machine learning and model serving. In an attempt to make TensorFlow more Java friendly, TensorFlow Java APIs were released in 2017, which enable scoring TensorFlow models using any Java Virtual Machine (JVM)–based language.

All of the aforementioned model export approaches are designed for platform-neutral descriptions of the models that need to be served. Introduction of these model export approaches led to the creation of several software products dedicated to “generic” model serving, for example, Openscoring and Open Data Group.

Another result of this standardization is the creation of open source projects, building generic “evaluators” based on these formats. JPMML and Hadrian are two examples that are being adopted more and more for building model-serving implementations, such as in these example projects: ING, R implementation, SparkML support, Flink support, and so on.

Additionally, because models are represented not as code but as data, usage of such a model description allows manipulation of models as a special type of data that is fundamental for our proposed solution.

## Why I Wrote This Book

This book describes the problem of serving models resulting from machine learning in streaming applications. It shows how to export trained models in TensorFlow and PMML formats and use them for model serving, using several popular streaming engines and frameworks.

I deliberately do not favor any specific solution. Instead, I outline options, with some pros and cons. The choice of the best solution depends greatly on the concrete use case that you are trying to solve, more precisely:

- The number of models to serve. Increasing the number of models will skew your preference toward the use of the key-based approach, like Flink key-based joins.
- The amount of data to be scored by each model. Increasing the volume of data suggests partition-based approaches, like Spark or Flink partition-based joins.
- The number of models that will be used to score each data item. You'll need a solution that easily supports the use of composite keys to match each data item to multiple models.
- The complexity of the calculations during scoring and additional processing of scored results. As the complexity grows, so will the load grow, which suggests using streaming engines rather than streaming libraries.
- Scalability requirements. If they are low, using streaming libraries like Akka and Kafka Streams can be a better option due to their relative simplicity compared to engines like Spark and Flink, their ease of adoption, and the relative ease of maintaining these applications.
- Your organization's existing expertise, which can suggest making choices that might be suboptimal, all other considerations being equal, but are more comfortable for your organization.

I hope this book provides the guidance you need for implementing your own solution.

## How This Book Is Organized

The book is organized as follows:

- Chapter 1 describes the overall proposed architecture.
- Chapter 2 talks about exporting models using examples of TensorFlow and PMML.
- Chapter 3 describes common components used in all solutions.
- Chapter 4 through Chapter 8 describe model serving implementations for different stream processing engines and frameworks.
- Chapter 9 covers monitoring approaches for model serving implementations.

## A Note About Code

The book contains a lot of code snippets. You can find the complete code in the following Git repositories:

- Python examples is the repository containing Python code for exporting TensorFlow models described in Chapter 2.
- Beam model server is the repository containing code for the Beam solution described in Chapter 5.
- Model serving is the repository containing the rest of the code described in the book.

## Acknowledgments

I would like to thank the people who helped me in writing this book and making it better, especially:

- Konrad Malawski, for his help with Akka implementation and overall review

- Dean Wampler, who did a thorough review of the overall book and provided many useful suggestions
- Trevor Grant, for conducting a technical review
- The entire Lightbend Fast Data team, especially Stavros Kontopoulos, Debasish Ghosh, and Jim Powers, for many useful comments and suggestions about the original text and code

# Chapter 1. Proposed Implementation

The majority of model serving implementations today are based on representational state transfer (REST), which might not be appropriate for high-volume data processing or for use in streaming systems. Using REST requires streaming applications to go “outside” of their execution environment and make an over-the-network call for obtaining model serving results.

The “native” implementation of new streaming engines—for example, Flink TensorFlow or Flink JPPML—do not have this problem but require that you restart the implementation to update the model because the model itself is part of the overall code implementation.

Here we present an architecture for scoring models natively in a streaming system that allows you to update models without interruption of execution.

## Overall Architecture

Figure 1-1 presents a high-level view of the proposed model serving architecture (similar to a dynamically controlled stream).

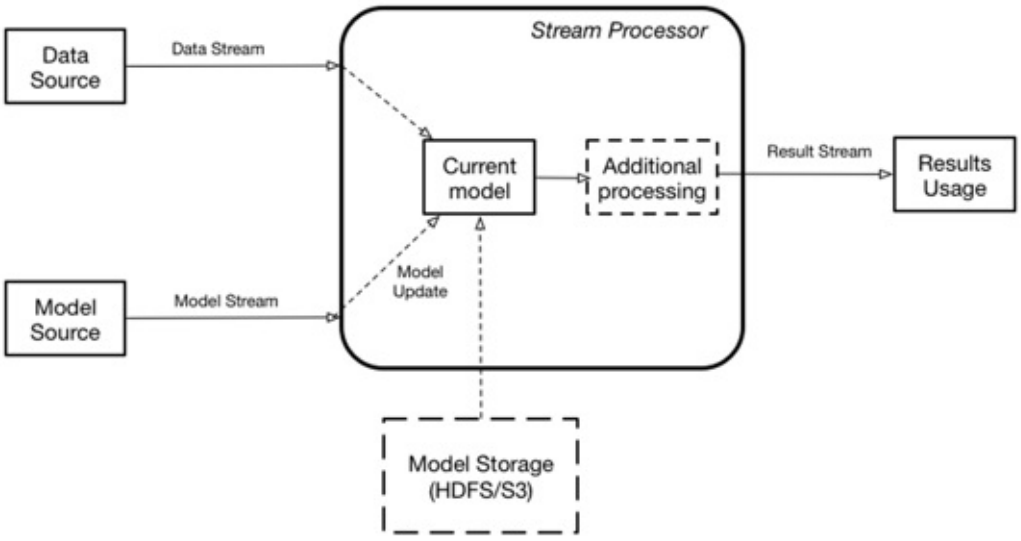



*Figure 1-1. Overall architecture of model serving*

This architecture assumes two data streams: one containing data that needs to be scored, and one containing the model updates. The streaming engine contains the current model used for the actual scoring in memory. The results of scoring can be either delivered to the customer or used by the streaming engine internally as a new stream—input for additional calculations. If there is no model currently defined, the input data is dropped. When the new model is received, it is instantiated in memory, and when instantiation is complete, scoring is switched to a new model. The model stream can either contain the binary blob of the data itself or the reference to the model data stored externally (pass by reference) in a database or a filesystem, like Hadoop Distributed File System (HDFS) or Amazon Web Services Simple Storage Service (S3).

Such approaches effectively using model scoring as a new type of functional transformation, which any other stream functional transformations can use.

Although the aforementioned overall architecture is showing a single model, a single streaming engine could score multiple models simultaneously.

## Model Learning Pipeline

For the longest period of time model building implementation was ad hoc—people would transform source data any way they saw fit, do some feature

extraction, and then train their models based on these features. The problem with this approach is that when someone wants to serve this model, he must discover all of those intermediate transformations and reimplement them in the serving application.

In an attempt to formalize this process, UC Berkeley AMPLab introduced the machine learning pipeline (Figure 1-2), which is a graph defining the complete chain of data transformation steps.

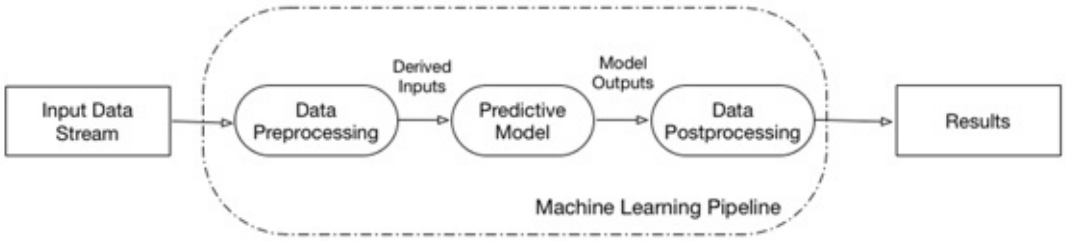



*Figure 1-2. The machine learning pipeline*

The advantage of this approach is twofold:

- It captures the entire processing pipeline, including data preparation transformations, machine learning itself, and any required postprocessing of the machine learning results. This means that the pipeline defines the complete transformation from well-defined inputs to outputs, thus simplifying update of the model.
- The definition of the complete pipeline allows for optimization of the processing.

A given pipeline can encapsulate more than one model (see, for example, PMML model composition). In this case, we consider such models internal—nonvisible for scoring. From a scoring point of view, a single pipeline always represents a single unit, regardless of how many models it encapsulates.

This notion of machine learning pipelines has been adopted by many applications including SparkML, TensorFlow, and PMML.

From this point forward in this book, when I refer to model serving, I mean serving the complete pipeline.

# Chapter 2. Exporting Models

Before delving into model serving, it is necessary to discuss the topic of exporting models. As discussed previously, data scientists define models, and engineers implement model serving. Hence, the ability to export models from data science tools is now important.

For this book, I will use two different examples: Predictive Model Markup Language (PMML) and TensorFlow. Let's look at the ways in which you can export models using these tools.

## TensorFlow

To facilitate easier implementation of model scoring, TensorFlow supports export of the trained models, which Java APIs can use to implement scoring. TensorFlow Java APIs are not doing the actual processing; they are just thin Java Native Interface (JNI) wrappers on top of the actual TensorFlow C++ code. Consequently, their usage requires "linking" the TensorFlow C++ executable to your Java application.

TensorFlow currently supports two types of model export: export of the execution graph, which can be optimized for inference, and a new SavedModel format, introduced this year.

## Exporting the Execution Graph

Exporting the execution graph is a "standard" TensorFlow approach to save the model. Let's take a look at an example of adding an execution graph export to a multiclass classification problem implementation using Keras with a TensorFlow backend applied to an open source wine quality dataset (complete code).

*Example 2-1. Exporting an execution graph from a Keras model*

```
...
# Create TF session and set it in Keras
sess = tf.Session()
K.set_session(sess)
```

```
...
# Saver op to save and restore all the variables
saver = tf.train.Saver()
# Save produced model
model_path = "path"
model_name = "WineQuality"
save_path = saver.save(sess, model_path+model_name+".ckpt")
print "Saved model at ", save_path
# Now freeze the graph (put variables into graph)
input_saver_def_path = ""
input_binary = False
output_node_names = "dense_3/Sigmoid"
restore_op_name = "save/restore_all"
filename_tensor_name = "save/Const:0"
output_frozen_graph_name = model_path + 'frozen_' + model_name
  + '.pb'
clear_devices = True
freeze_graph.freeze_graph(graph_path, input_saver_def_path,
input_binary, save_path,Output_node_names,
restore_op_name, filename_tensor_name,
output_frozen_graph_name, clear_devices, "")
# Optimizing graph
input_graph_def = tf.GraphDef()
with tf.gfile.Open(output_frozen_graph_name, "r") as f:
   data = f.read()
   input_graph_def.ParseFromString(data)
output_graph_def =
  optimize_for_inference_lib.optimize_for_inference(
   input_graph_def,
   ["dense_1_input"],
   ["dense_3/Sigmoid"],
   tf.float32.as_datatype_enum)
# Save the optimized graph
tf.train.write_graph(output_graph_def, model_path,
  "optimized_" + model_name + ".pb", as_text=False)
```

Example 2-1 is adapted from a Keras machine learning example to demonstrate how to export a TensorFlow graph. To do this, it is necessary to explicitly set the TensorFlow session for Keras execution. The TensorFlow execution graph is tied to the execution session, so the session is required to gain access to the graph.

The actual graph export implementation involves the following steps:

1. Save initial graph.
2. Freeze the graph (this means merging the graph definition with parameters).
3. Optimize the graph for serving (remove elements that do not affect

serving).

4. Save the optimized graph.

The saved graph is an optimized graph stored using the binary Google protocol buffer (protobuf) format, which contains only portions of the overall graph and data relevant for model serving (the portions of the graph implementing learning and intermediate calculations are dropped).

After the model is exported, you can use it for scoring. Example 2-2 uses the TensorFlow Java APIs to load and score the model (full code available here).

*Example 2-2. Serving the model created from the execution graph of the Keras model*

```
class WineModelServing(path : String) {
 import WineModelServing._
 // Constructor
 val lg = readGraph(Paths.get (path))
 val ls = new Session (lg)

 def score(record : Array[Float]) : Double = {
   val input = Tensor.create(Array(record))
   val result = ls.runner.feed("dense_1_input",input).
     fetch("dense_3/Sigmoid").run().get(0)
   // Extract result value
   val rshape = result.shape
   var rMatrix =
     Array.ofDim[Float](rshape(0).asInstanceOf[Int],rshape(1).
       asInstanceOf[Int])result.copyTo(rMatrix)
   var value = (0, rMatrix(0)(0))
   1 to (rshape(1).asInstanceOf[Int] -1) foreach{i => {
      if(rMatrix(0)(i) > value._2)
        value = (i, rMatrix(0)(i))
    }}
    value._1.toDouble
 }
 def cleanup() : Unit = {
   ls.close
 }
}
object WineModelServing{
 def main(args: Array[String]): Unit = {
   val model_path = "/optimized_WineQuality.pb"   // model
   val data_path = "/winequality_red.csv" // data
   val lmodel = new WineModelServing(model_path)
   val inputs = getListOfRecords(data_path)
   inputs.foreach(record =>
```

```
    println(s"result ${lmodel.score(record._1)}"))
  lmodel.cleanup()
 }
 private def readGraph(path: Path) : Graph = {
  try {
    val graphData = Files.readAllBytes(path)
    val g = new Graph
    g.importGraphDef(graphData)
    g
  } ...
 }
...
}
```

In this simple code, the constructor uses the `readGraph` method to read the execution graph and create a TensorFlow session with this graph attached to it.

The score method takes an input record containing wine quality observations and converts it to a tensor format, which is used as an input to the running graph. Because the exported graph does not provide any information about names and shapes of either inputs or outputs (the execution signature), when using this approach, it is necessary to know which variable(s) (i.e., input parameter) your flow accepts (feed) and which tensor(s) (and their shape) to fetch as a result. After the result is received (in the form of a tensor), its value is extracted.

The execution is orchestrated by the main method in the `WineModelServing` object. This method first creates an instance of the `WineModelServing` class and then reads the list of input records and for each record invokes a serve method on the `WineModelServing` class instance.

To run this code, in addition to the TensorFlow Java library, you must also have the TensorFlow C++ implementation library (*.dll* or *.so*) installed on the machine that will run the code.

Advantages of execution graph export include the following:

- Due to the optimizations, the exported graph has a relatively small size.
- The model is self-contained in a single file, which makes it easy to transport it as a binary blob, for instance, using a Kafka topic.

A disadvantage is that the user of the model must know explicitly both input and output (and their shape and type) of the model to use the graph correctly;

however, this is typically not a serious problem.

## Exporting the Saved Model

TensorFlow *SavedModel* is a new export format, introduced in 2017, in which the model is exported as a directory with the following structure:

```
assets/
assets.extra/
variables/
    variables.data-?????-of-?????
    variables.index
Saved_model.pb
```

where:

- `assets` is a subfolder containing auxiliary files such as vocabularies, etc.
- `assets.extra` is a subfolder where higher-level libraries and users can add their own assets that coexist with the model but are not loaded by the graph. It is not managed by the SavedModel libraries.
- `variables` is a subfolder containing output from the TensorFlow Saver: both variables index and data.
- `saved_model.pb` contains graph and MetaGraph definitions in binary protocol buffer format.

The advantages of the SavedModel format are:

- You can add multiple graphs sharing a single set of variables and assets to a single SavedModel. Each graph is associated with a specific set of tags to allow identification during a load or restore operation.
- Support for SignatureDefs. The definition of graph inputs and outputs (including shape and type for each of them) is called a Signature. SavedModel uses SignatureDefs to allow generic support for signatures that might need to be saved with the graphs.
- Support for assets. In some cases, TensorFlow operations depend on external files for initialization, for example, vocabularies. SavedModel

exports these additional files in the assets directory.

Here is a Python code snippet (complete code available here) that shows you how to save a trained model in a saved model format:

*Example 2-3. Exporting saved model from a Keras model*

```
#export_version =...  # version number (integer)
export_dir = "savedmodels/WineQuality/"
builder = saved_model_builder.SavedModelBuilder(export_dir)
signature = predict_signature_def(inputs={'winedata': model.input},
  outputs={'quality': model.output})
builder.add_meta_graph_and_variables(sess=sess,
  tags=[tag_constants.SERVING],
  signature_def_map={'predict': signature})
builder.save()
```

By replacing the export execution graph in Example 2-1 with this code, it is possible to get a saved model from your multiclass classification problem.

After you export the model into a directory, you can use it for serving. Example 2-4 (complete code available here) takes advantage of the TensorFlow Java APIs to load and score with the model.

*Example 2-4. Serving a model based on the saved model from a Keras model*

```
object WineModelServingBundle {
 def apply(path: String, label: String): WineModelServingBundle =
new WineModelServingBundle(path, label)
 def main(args: Array[String]): Unit = {
   val data_path = "/winequality_red.csv"
   val saved_model_path = "/savedmodels/WineQuality"
   val label = "serve"
   val model = WineModelServingBundle(saved_model_path, label)
   val inputs = getListOfRecords(data_path)
   inputs.foreach(record =>
     println(s"result ${model.score(record._1)}
       expected ${record._2}"))
   model.cleanup()
 }
...
class WineModelServingBundle(path : String, label : String){
 val bundle = SavedModelBundle.load(path, label)
 val ls: Session = bundle.session
 val metaGraphDef = MetaGraphDef.parseFrom(bundle.metaGraphDef())
 val signatures = parseSignature(
   metaGraphDef.getSignatureDefMap.asScala)
 def score(record : Array[Float]) : Double = {
   val input = Tensor.create(Array(record))
```

```
   val result = ls.runner.feed(signatures(0).inputs(0).name, input)
 .fetch(signatures(0).outputs(0).name).run().get(0)
   ...
 }
...
 def convertParameters(tensorInfo: Map[String,TensorInfo]) :
   Seq[Parameter] = {
   var parameters = Seq.empty[Parameter]
   tensorInfo.foreach(input => {
     ...
     fields.foreach(descriptor => {
       descriptor._2.asInstanceOf[TensorShapeProto].getDimList
      .toArray.map(d => d.asInstanceOf[
        TensorShapeProto.Dim].getSize)
      .toSeq.foreach(v => shape = shape :+ v.toInt)
      .foreach(v => shape = shape :+ v.toInt)
       }
       if(descriptor._1.getName.contains("name") ) {
         name = descriptor._2.toString.split(":")(0)
       }
       if(descriptor._1.getName.contains("dtype") ) {
         dtype = descriptor._2.toString
       }
     })
     parameters = Parameter(name, dtype, shape) +: parameters
   })
   parameters
 }
 def parseSignature(signatureMap : Map[String, SignatureDef])
   : Seq[Signature] = {

   var signatures = Seq.empty[Signature]
   signatureMap.foreach(definition => {
     val inputDefs = definition._2.getInputsMap.asScala
     val outputDefs = definition._2.getOutputsMap.asScala
     val inputs = convertParameters(inputDefs)
     val outputs = convertParameters(outputDefs)
     signatures = Signature(definition._1, inputs, outputs)
       +: signatures
   })
   signatures
 }
}
...
```

Compare this code with the code in Example 2-2. Although the main structure is the same, there are two significant differences:

- Reading the graph is more involved. The saved model contains not just the

graph itself, but the entire bundle (directory), and then obtains the graph from the bundle. Additionally, it is possible to extract the method signature (as a protobuf definition) and parse it to get inputs and output for your method of execution. Keep in mind that, in general, the graph read from the bundle can contain multiple signatures, so it is necessary to pick the appropriate signature by name. This name is defined during model saving (*winedata*, defined in Example 2-3). In the code, because I know that there is only one signature, I just took the first element of the array.

- In the implementation method, instead of hardcoding names of inputs and outputs, I rely on the signature definition.

> **WARNING**
>
> When saving parameter names, TensorFlow uses the convention *name:column*. For example, in our case the inputs name, *dense_1_input*, with a single column (0) is represented as *dense_1_input:0*. The Java APIs do not support this notation, so the code splits the name at ":" and returns only the first substring.

Additionally, there is currently work underway to convert TensorFlow exported models (in the saved models format) to PMML. When this work is complete, developers will have additional choices for building scoring solutions for models exported from TensorFlow.

## PMML

In our next example, Random Forest Classifier, using the same wine quality dataset that was used in the multiclass classification with the TensorFlow example, we show how to use JPMML/SparkML for exporting models from SparkML machine learning. The code looks as shown in Example 2-5 (complete code available here).

*Example 2-5. Random Forest Classifier using SparkML with PMML export*

```
object WineQualityRandomForestClassifierPMML {
 def main(args: Array[String]): Unit = {
   ...
   // Load and parse the data file
```

```
   ...
   // Decision Tree operates on feature vectors
   val assembler = new VectorAssembler().
     setInputCols(inputFields.toArray).setOutputCol("features")
   // Fit on whole dataset to include all labels in index.
   val labelIndexer = new StringIndexer()
     .setInputCol("quality").setOutputCol("indexedLabel").fit(dff)
   // Create classifier
   val dt = new RandomForestClassifier().setLabelCol("indexedLabel")
      .setFeaturesCol("features").setNumTrees(10)
   // Convert indexed labels back to original labels.
   val labelConverter= new IndexToString().setInputCol("prediction")
    .setOutputCol("predictedLabel").setLabels(labelIndexer.labels)
   // Create pileline
   val pipeline = new Pipeline()
     .setStages(Array(assembler, labelIndexer, dt, labelConverter))
   // Train model
   val model = pipeline.fit(dff)
   // PMML
   val schema = dff.schema
   val pmml = ConverterUtil.toPMML(schema, model)
   MetroJAXBUtil.marshalPMML(pmml, System.out)
   spark.stop()
 }
}
```

The bulk of the code defines the machine learning pipeline, which contains the following components:

Vector assembler

: A transformer that combines a given list of columns into a single vector column.

Label indexer

: This encodes a string column of labels to a column of label indices.

Classifier

: A Random Forest classifier is used here, which belongs to a popular family of classification and regression methods.

Label converter

: This maps a column of label indices back to a column containing the original labels as strings.

After the pipeline is built, it is trained, and then the PMML exporter uses a data frame schema and the pipeline definition to export the complete pipeline, with parameters, in PMML format.

After you export the model, you can use it for scoring. Example 2-6 uses the JPMML evaluator library to load and score the model (complete code available here).

*Example 2-6. Serving PMML model*

```
class WineQualityRandomForestClassifier(path : String) {
 import WineQualityRandomForestClassifier._
 var arguments = mutable.Map[FieldName, FieldValue]()
 // constructor
 val pmml: PMML = readPMML(path)
 optimize(pmml)
 // Create and verify evaluator
 val evaluator = ModelEvaluatorFactory.newInstance()
   .newModelEvaluator(pmml)
 evaluator.verify()
 // Get input/target fields
 val inputFields = evaluator.getInputFields
 val target: TargetField = evaluator.getTargetFields.get(0)
 val tname = target.getName

 def score(record : Array[Float]) : Double = {
   arguments.clear()
   inputFields.foreach(field => {
     arguments.put(field.getName, field
       .prepare(getValueByName(record, field.getName.getValue)))
   })
   // Calculate output
   val result = evaluator.evaluate(arguments)
   // Convert output
   result.get(tname) match {
      case c : Computable => c.getResult.toString.toDouble
      case v : Any => v.asInstanceOf[Double]
   }
 }
...
object WineQualityRandomForestClassifier {
 def main(args: Array[String]): Unit = {
   val model_path = "data/
     winequalityRandonForrestClassification.pmml"
   val data_path = "data/winequality_red.csv"
   val lmodel = new WineQualityRandomForestClassifier(model_path)
   val inputs = getListOfRecords(data_path)
   inputs.foreach(record =>
```

```
    println(s"result ${lmodel.score(record._1)}"))
 }
 def readPMML(file: String): PMML = {
   var is = null.asInstanceOf[InputStream]
   try {
     is = new FileInputStream(file)
     PMMLUtil.unmarshal(is)
   }
   finally if (is != null) is.close()
 }
 private val optimizers = ...
 def optimize(pmml : PMML) = this.synchronized {
   ...
 }
...
}
```

In this simple example, the constructor calls the `readPMML` method to read the PMML model and then invokes the optimize method. We use the optimized PMML (optimizers change default generation to allow for more efficient execution) representation that is returned to create the evaluator.

The score method takes an input record containing quality observations and converts them to the format acceptable by evaluator. Then, data is passed to the evaluator to produce a score. Finally, an actual value is extracted from the result.

The execution is orchestrated by the main method in the `WineQualityRandomForestClassifier` object. This method first creates an instance of the `WineQualityRandomForestClassifier` class and then reads the list of input records and invokes the serve method in the `WineQualityRandomForestClassifier` class instance on each record.

Now that we know how to export models, let’s discuss how you can use these models for actual model scoring.

# Chapter 3. Implementing Model Scoring

As depicted in Figure 1-1, the overall architecture of our implementation is reading two input streams, the models stream and the data stream, and then joining them for producing the results stream.

There are two main options today for implementing such stream-based applications: stream-processing engines or stream-processing libraries:

- Modern stream-processing engines (SPEs) take advantage of cluster architectures. They organize computations into a set of operators, which enables execution parallelism; different operators can run on different threads or different machines. An engine manages operator distribution among the cluster's machines. Additionally, SPEs typically implement checkpointing, which allows seamless restart execution in case of failures.
- A stream-processing library (SPL), on the other hand, is a library, and often domain-specific language (DSL), of constructs simplifying building streaming applications. Such libraries typically do not support distribution and/or clustering; this is typically left as an exercise for developer.

Although these are very different, because both have the word "stream" in them, they are often used interchangeably. In reality, as outlined in Jay Kreps's blog, they are two very different approaches to building streaming applications, and choosing one of them is a trade-off between power and simplicity. A side-by-side comparison of Flink and Kafka Streams outlines the major differences between the two approaches, which lies in

> *the way they are deployed and managed (which often is a function of who owns these applications from an organizational perspective) and how the parallel processing (including fault tolerance) is coordinated. These are core differences—they are ingrained in the architecture of these two approaches.*

Using an SPE is a good fit for applications that require features provided out of

the box by such engines, including scalability and high throughput through parallelism across a cluster, event-time semantics, checkpointing, built-in support for monitoring and management, and mixing of stream and batch processing. The drawback of using engines is that you are constrained with the programming and deployment models they provide.

In contrast, SPLs provide a programming model that allows developers to build the applications or microservices the way that fits their precise needs and deploy them as simple standalone Java applications. But in this case, developers need to roll out their own scalability, high availability, and monitoring solutions (Kafka Streams supports some of them by using Kafka).

Today's most popular SPEs includes: Apache Spark, Apache Flink, Apache Beam, whereas most popular stream libraries are Apache Kafka Streams and Akka Streams. In the following chapters, I show how you can use each of them to implement our architecture of model serving.

There are several common artifacts in my implementations that are used regardless of the streaming engine/framework: model representation, model stream, data stream, model factory, and test harness, all of which are described in the following sections.

## Model Representation

Before diving into specific implementation details, you must decide on the model's representation. The question here is whether it is necessary to introduce special abstractions to simplify usage of the model in specific streaming libraries.

I decided to represent model serving as an "ordinary" function that can be used at any place of the stream processing pipeline. Additionally, representation of the model as a simple function allows for a functional composition of models, which makes it simpler to combine multiple models for processing. Also, comparison of Examples 2-2, 2-4, and 2-6, shows that different model types (PMML versus TensorFlow) and different representations (saved model versus ordinary graph) result in the same basic structure of the model scoring pipeline that can be generically described using the Scala trait shown in Example 3-1.

*Example 3-1. Model representation*

```
trait Model {
 def score(input : AnyVal) : AnyVal
 def cleanup() : Unit
 def toBytes() : Array[Byte]
 def getType : Long
}
```

The basic methods of this trait are as follows:

- `score()` is the basic method for model implementation, converting input data into a result or score.
- `cleanup()` is a hook for a model implementer to release all of the resources associated with the model execution-model lifecycle support.
- `toBytes()` is a supporting method used for serialization of the model content (used for checkpointing).
- `getType()` is a supporting method returning an index for the type of model used for finding the appropriate model factory class (see the section that follows).

This trait can be implemented using JPMML or TensorFlow Java APIs and used at any place where model scoring is required.

## Model Stream

It is also necessary to define a format for representing models in the stream. I decided to use Google protocol buffers ("protobuf" for short) for model representation, as demonstrated in Example 3-2.

*Example 3-2. Protobuf definition for the model update*

```
syntax = "proto3";
// Description of the trained model
message ModelDescriptor {
   // Model name
   string name = 1;
   // Human-readable description
   string description = 2;
   // Data type for which this model is applied
   string dataType = 3;
```

```
    // Model type
    enum ModelType {
        TENSORFLOW  = 0;
        TENSORFLOWSAVED = 1;
        PMML = 2;
    };
    ModelType modeltype = 4;
    oneof MessageContent {
        bytes data = 5;
        string location = 6;
    }
}
```

The model here (model content) can be represented either inline as a byte array or as a reference to a location where the model is stored. In addition to the model data our definition contains the following fields:

`name`

: The name of the model that can be used for monitoring or UI applications.

`description`

: A human-readable model description that you can use for UI applications.

`dataType`

: Data type for which this model is applicable (our model stream can contain multiple different models, each used for specific data in the stream). See Chapter 4 for more details of this field utilization.

`modelType`

: For now, I define only three model types, PMML and two TensorFlow types, graph and saved models.

Throughout implementation, ScalaPB is used for protobuf marshaling, generation, and processing.

## Data Stream

Similar to the model stream, protobufs are used for the data feed definition and encoding. Obviously, a specific definition depends on the actual data stream that you are working with. For our wine quality dataset, the protobuf looks like

Example 3-3.

*Example 3-3. Protobuf definition for the data feed*

```
syntax = "proto3";
// Description of the wine parameters
message WineRecord {
   double fixed_acidity = 1;
   double volatile_acidity = 2;
   double citric_acid = 3;
   double residual_sugar = 4;
   double chlorides = 5;
   double free_sulfur_dioxide = 6;
   double total_sulfur_dioxide = 7;
   double density = 8;
   double pH = 9;
   double sulphates = 10;
   double alcohol = 11;
   // Data type for this record
   string dataType = 12;
}
```

Note that a `dataType` field is added to both the model and data definitions. This field is used to identify the record type (in the case of multiple data types and models) to match it to the corresponding models.

In this simple case, I am using only a single concrete data type, so Example 3-3 shows direct data encoding. If it is necessary to support multiple data types, you can either use protobuf's `oneof` construct, if all the records are coming through the same stream, or separate streams, managed using separate Kafka topics, can be introduced, one for each data type.

The proposed data type–based linkage between data and model feeds works well when a given record is scored with a single model. If this relationship is one-to-many, where each record needs to be scored by multiple models, a composite key (data type with model ID) can be generated for every received record.

## Model Factory

As defined above, a model in the stream is delivered in the form of a protobuf message that can contain either a complete representation of the model or a reference to the model location. To generalize model creation from the protobuf message, I introduce an additional trait, `ModelFactory`, which supports building

models out of a Model Descriptor (an internal representation of the protobuf definition for the model update [in Example 3-2]; the actual code for this class is shown later in the book). An additional use for this interface is to support serialization and deserialization for checkpointing. We can describe the model factory using the trait presented in Example 3-4.

*Example 3-4. Model factory representation*

```
trait ModelFactory {
 def create(input : ModelDescriptor) : Model
 def restore(bytes : Array[Byte]) : Model
}
```

Here are the basic methods of this trait:

`create`

A method creating a model based on the Model descriptor.

`restore`

A method to restore a model from a byte array emitted by the model's `toByte` method. These two methods need to cooperate to ensure proper functionality of `ModelSerializer`/`ModelDeserializer`.

## Test Harness

I have also created a small test harness to be able to validate implementations. It comprises two simple Kafka clients: one for models, one for data. Both are based on a common `KafkaMessageSender` class (complete code available here) shown in Example 3-5.

*Example 3-5. The KafkaMessageSender class*

```
class KafkaMessageSender (brokers: String, zookeeper : String){

 // Configure
 ...
 // Create producer
 val producer = new KafkaProducer[Array[Byte], Array[Byte]](props)
 val zkUtils = ZkUtils.apply(zookeeper, KafkaMessageSender
   .sessionTimeout, KafkaMessageSender.connectionTimeout, false)

 // Write value to the queue
 def writeValue(topic:String, value:Array[Byte]): RecordMetadata = {
```

```
    val result = producer.send(
      new ProducerRecord[Array[Byte],
        Array[Byte]](topic, null, value)).get
        producer.flush()
    result
  }

  // Close producer
  def close(): Unit = {
    producer.close
  }

  def createTopic(topic : String, numPartitions: Int = 1,
    replicationFactor : Int = 1): Unit = {
    if (!AdminUtils.topicExists(zkUtils, topic)){
      try {
        AdminUtils.createTopic(...
      }catch {
        ...
      }
    }
    else
      println(s"Topic $topic already exists")
  }
}

object KafkaMessageSender{
  ...
  private val senders : Map[String, KafkaMessageSender] = Map()

  def apply(brokers:String, zookeeper :String): KafkaMessageSender ={
    senders.get(brokers) match {
      case Some(sender) => sender
      case _ => {
        val sender = new KafkaMessageSender(brokers, zookeeper)
        senders.put(brokers, sender)
        sender
      }
    }
  }
}
```

This class uses Kafka APIs to create a Kafka producer and send messages. The `DataProvider` class uses the `KafkaMessageSender` class to send data messages (complete code available here), as demonstrated in Example 3-6.

*Example 3-6. DataProvider class*

```
DataProvider {
```

```
 ...

 def main(args: Array[String]) {
   val sender = KafkaMessageSender(
     ApplicationKafkaParameters.LOCAL_KAFKA_BROKER,
     ApplicationKafkaParameters.LOCAL_ZOOKEEPER_HOST)
   sender.createTopic(ApplicationKafkaParameters.DATA_TOPIC)
   val bos = new ByteArrayOutputStream()
   val records  = getListOfRecords(file)
   while (true) {
     var lineCounter = 0
     records.foreach(r => {
       bos.reset()
       r.writeTo(bos)
       sender.writeValue(ApplicationKafkaParameters.DATA_TOPIC,
         bos.toByteArray)
       pause()
     })
     pause()
   }
 }
 ...
 def getListOfRecords(file: String): Seq[WineRecord] = {

   var result = Seq.empty[WineRecord]
   val bufferedSource = Source.fromFile(file)
   for (line <- bufferedSource.getLines) {
     val cols = line.split(";").map(_.trim)
     val record = new WineRecord(
       fixedAcidity = cols(0).toDouble,
       volatileAcidity = cols(1).toDouble,
       citricAcid = cols(2).toDouble,
       residualSugar = cols(3).toDouble,
       chlorides = cols(4).toDouble,
       freeSulfurDioxide = cols(5).toDouble,
       totalSulfurDioxide = cols(6).toDouble,
       density = cols(7).toDouble,
       pH = cols(8).toDouble,
       sulphates = cols(9).toDouble,
       alcohol = cols(10).toDouble,
       dataType = "wine"
     )
     result = record +: result
   }
   bufferedSource.close
   result
 }
}
```

The actual data is the same data as that used for training the wine quality dataset, which is stored locally in the form of a CSV file. The file is read into memory and then the records are converted from text records to protobuf records, which are published to the Kafka topic using an infinite loop with a predefined pause between sends.

A similar implementation produces models for serving. For the set of models, I am using results of different training algorithms in both TensorFlow (exported as execution graph) and PMML formats, which are published to the Kafka topic using an infinite loop with a predefined pause between sends.

Now that we have outlined the necessary components, Chapter 4 through Chapter 8 demonstrate how you can implement this solution using specific technology.

# Chapter 4. Apache Flink Implementation

Flink is an open source stream-processing engine (SPE) that does the following:

- Scales well, running on thousands of nodes
- Provides powerful checkpointing and save pointing facilities that enable fault tolerance and restartability
- Provides state support for streaming applications, which allows minimization of usage of external databases for streaming applications
- Provides powerful window semantics, allowing you to produce accurate results, even in the case of out-of-order or late-arriving data

Let's take a look how we can use Flink's capabilities to implement the proposed architecture

## Overall Architecture

Flink provides a low-level stream processing operation, `ProcessFunction`, which provides access to the basic building blocks of any streaming application:

- Events (individual records within a stream)
- State (fault-tolerant, consistent)
- Timers (event time and processing time)

Implementation of low-level operations on two input streams is provided by Flink's low-level join operation, which is bound to two different inputs (if we need to merge more than two streams it is possible to cascade multiple low-level joins; additionally side inputs, scheduled for the upcoming versions of Flink, would allow additional approaches to stream merging) and provides individual methods for processing records from each input. Implementing a low-level join

typically follows the following pattern:

1. Create and maintain a state object reflecting the current state of execution.
2. Update the state upon receiving elements from one (or both) input(s).
3. Upon receiving elements from one or both input(s) use the current state to transform data and produce the result.

Figure 4-1 illustrates this operation.

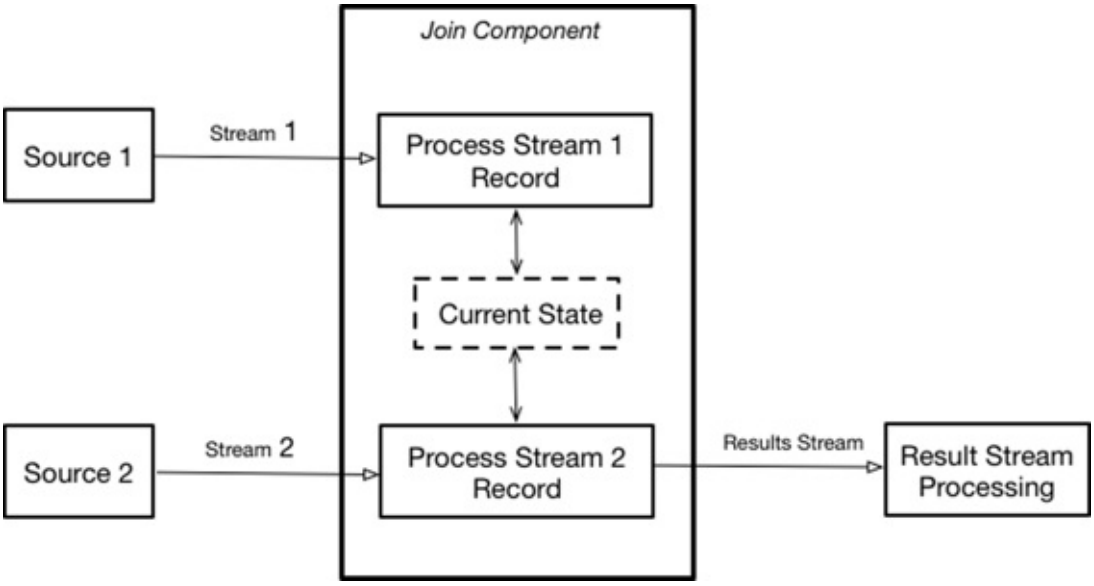



*Figure 4-1. Using Flink's low-level join*

This pattern fits well into the overall architecture (Figure 1-1), which is what I want to implement.

Flink provides two ways of implementing low-level joins, key-based joins implemented by `CoProcessFunction`, and partition-based joins implemented by `RichCoFlatMapFunction`. Although you can use both for this implementation, they provide different service-level agreements (SLAs) and are applicable for slightly different use cases.

## Using Key-Based Joins

Flink's `CoProcessFunction` allows key-based merging of two streams. When using this API, data is partitioned by key across multiple Flink executors.

Records from both streams are routed (based on key) to an appropriate executor that is responsible for the actual processing, as illustrated in Figure 4-2.

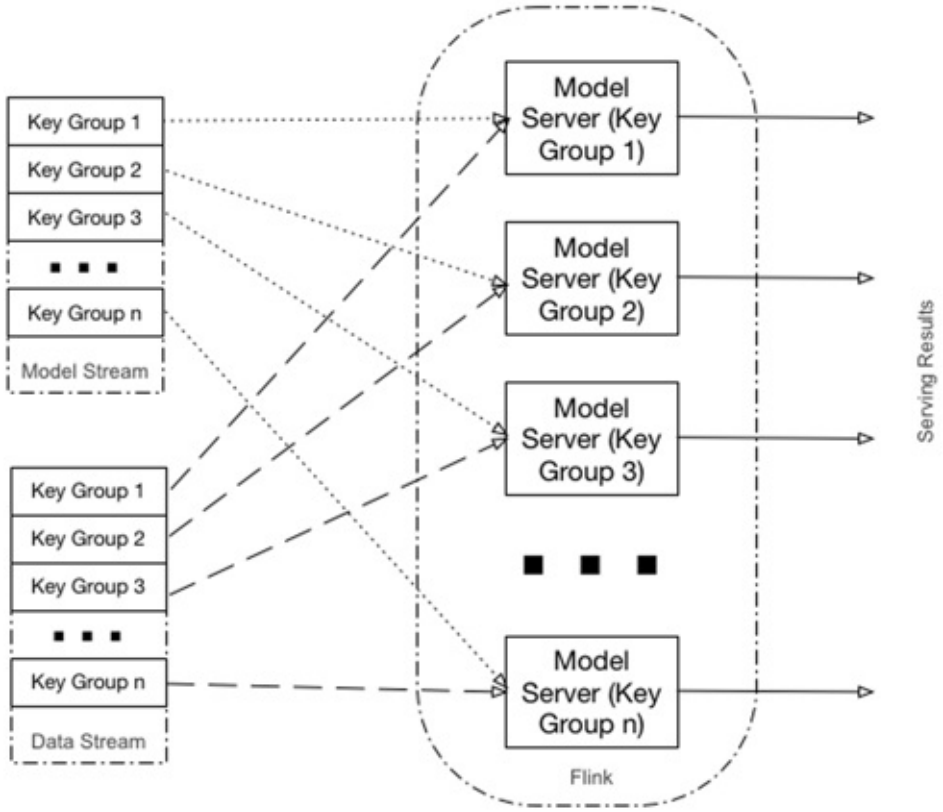



*Figure 4-2. Key-based join*

Here are the main characteristics of this approach:

- Distribution of execution is based on key (`dataType`, see Examples 3-2 and 3-3).
- Individual models' scoring (for a given `dataType`) is implemented by a separate executor (a single executor can score multiple models), which means that scaling Flink leads to a better distribution of individual models and consequently better parallelization of scorings.
- A given model is always scored by a given executor, which means that depending on the data type distribution of input records, this approach can lead to "hot" executors

Based on this, key-based joins are an appropriate approach for the situations

when it is necessary to score multiple data types with relatively even distribution.

In the heart of this implementation is a `DataProcessor` class (complete code available here), which you can see in Example 4-1.

*Example 4-1. The DataProcessor class*

```
object DataProcessorKeyed {
 def apply() = new DataProcessorKeyed
...
}
class DataProcessor extends
CoProcessFunction[WineRecord, ModelToServe, Double]
with CheckpointedFunction
with CheckpointedRestoring[List[Option[Model]]] {


 var currentModel : Option[Model] = None
 var newModel : Option[Model] = None
 @transient private var checkpointedState:
   ListState[Option[Model]] = null

override def
  snapshotState(context: FunctionSnapshotContext): Unit = {
 checkpointedState.clear()
 checkpointedState.add(currentModel)
 checkpointedState.add(newModel)
}

override def initializeState(context:
  FunctionInitializationContext): Unit = {
 val descriptor = new ListStateDescriptor[Option[Model]] (
   "modelState",new ModelTypeSerializer)

 checkpointedState = context.getOperatorStateStore.
   getListState (descriptor)

 if (context.isRestored) {
   val iterator = checkpointedState.get().iterator()
   currentModel = iterator.next()
   newModel = iterator.next()
 }
}

override def restoreState(state: List[Option[Model]]): Unit = {
 currentModel = state(0)
 newModel = state(1)
}
```

```
 override def processElement2(model: ModelToServe, ctx:
   CoProcessFunction[WineRecord,ModelToServe, Double]#Context,
   out: Collector[Double]): Unit = {

 import DataProcessorKeyed._

 println(s"New model - $model")
 newModel = factories.get(model.modelType) match {
   case Some(factory) => factory.create (model)
   case _ => None
 }
}

 override def processElement1(record: WineRecord, ctx:
   CoProcessFunction[WineRecord,ModelToServe, Double]#Context,
   out: Collector[Double]): Unit = {


   // See if we have update for the model
   newModel match {
     case Some(model) => {
       // Clean up current model
       currentModel match {
         case Some(m) => m.cleanup()
         case _ =>
       }
       // Update model
       currentModel = Some(model)
       newModel = None
     }
     case _ =>
   }
   currentModel match {
     case Some(model) => {
       val start = System.currentTimeMillis()
       val quality = model.score(record.asInstanceOf[AnyVal]).
         asInstanceOf[Double]
       val duration = System.currentTimeMillis() - start
       modelState.update(modelState.value()
         .incrementUsage(duration))
       println(s"Calculated quality - $quality")
     }
     case _ => println("No model available - skipping")
   }
 }
}
```

This class has two main methods:

`processElement2`

: This method is invoked when a new `Model` record (`ModelToServe` class, described later) arrives. This method just builds a new model to serve, as in Example 2-2 for TensorFlow models or Example 2-6 for PMML models, and stores it in a `newModel` state variable. Because model creation can be a lengthy operation, I am separating a `newModel` state from a `currentModel` state, so that model creation does not affect current model serving.

`processElement1`

: This is invoked when a new `Data` record (`WineRecord` class) arrives. Here, the availability of a new model is first checked, and if it is available, the `currentModel` is updated with the value of `newModel`. This ensures that the model update will never occur while scoring a record. We then check whether there is currently a model to score and invoke the actual scoring.

In addition to these main methods, the class also implements support for checkpointing of managed state. We do this by adding two additional interfaces to the class:

`CheckpointedFunction`

: The core interface for stateful transformation functions that maintain state across individual stream records.

`CheckpointedRestoring`

: The interface providing methods for restoring state from the checkpointing.

These two interfaces are implemented by the following three methods: `initializeState`, `snapshotState`, and `restoreState`.

Example 4-2 shows what the `ModelToServe` class used by the `DataProcessor` class looks like.

*Example 4-2. The ModelToServe class*

```
object ModelToServe {
 def fromByteArray(message: Array[Byte]): Try[ModelToServe] = Try{
   val m = ModelDescriptor.parseFrom(message)
   m.messageContent.isData match {
     case true => new ModelToServe(m.name, m.description,
```

```
      m.modeltype, m.getData.toByteArray, m.dataType)
    case _ => throw new Exception("Not yet supported")
  }
 }
}

case class ModelToServe(name: String, description: String,
  modelType: ModelDescriptor.ModelType,
  model : Array[Byte], dataType : String) {}
```

This class unmarshals incoming protobufs of the model definition (Example 3-2) and converts it into the internal format used in the rest of the code.

Similarly, we use the `DataRecord` class to unmarshal the incoming data definition (Example 3-1), as demonstrated in Example 4-3.

*Example 4-3. The DataRecord class*

```
object DataRecord {

 def fromByteArray(message: Array[Byte]): Try[WineRecord] = Try {
   WineRecord.parseFrom(message)
 }
}
```

Implementation of checkpointing also requires serialization support for the `Model` class, shown in Example 4-4 (complete code available here).

*Example 4-4. The ModelTypeSerializer class*

```
class ModelTypeSerializer extends TypeSerializer[Option[Model]] {
 ...

 override def serialize(record: Option[Model],
   target: DataOutputView): Unit = {
   record match {
     case Some(model) => {
       target.writeBoolean(true)
       val content = model.toBytes()
       target.writeLong(model.getType)
       target.writeLong(content.length)
       target.write(content)
     }
     case _ => target.writeBoolean(false)
   }
 }

 ...
```

```
 override def deserialize(source: DataInputView): Option[Model] =
   source.readBoolean() match {
     case true => {
       val t = source.readLong().asInstanceOf[Int]
       val size = source.readLong().asInstanceOf[Int]
       val content = new Array[Byte] (size)
       source.read (content)
       Some(factories.get(t).get.restore(content))
     }
     case _ => None
   }
...
```

This class leverages utility methods on the model and model factory traits to generically implement serialization/deserialization regardless of the actual model implementation.

Serialization implementation also requires implementation of configuration support, which you can see in Example 4-5 (complete code available here).

*Example 4-5. The ModelSerializerConfigSnapshot class*

```
class ModelSerializerConfigSnapshot[T <: Model]
  extends TypeSerializerConfigSnapshot{

...

 override def write(out: DataOutputView): Unit = {
   super.write(out)
   // write only the classname to avoid Java serialization
   out.writeUTF(classOf[Model].getName)
 }

 override def read(in: DataInputView): Unit = {
   super.read(in)
   val genericTypeClassname = in.readUTF
   try
     typeClass = Class.forName(genericTypeClassname, true,
       getUserCodeClassLoader).asInstanceOf[Class[Model]]
   catch {
     ...
   }
 }
...
```

Overall orchestration of the execution is done using a Flink driver, shown in Example 4-6 (complete code available here).

*Example 4-6. Flink driver for key-based joins*

```
object ModelServingKeyedJob {
...
 // Build execution Graph
 def buildGraph(env : StreamExecutionEnvironment) : Unit = {
   env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
   env.enableCheckpointing(5000)
   // Configure Kafka consumer
   val dataKafkaProps = new Properties
   dataKafkaProps.setProperty("zookeeper.connect",
     ModelServingConfiguration.LOCAL_ZOOKEEPER_HOST)
   dataKafkaProps.setProperty("bootstrap.servers",
     ModelServingConfiguration.LOCAL_KAFKA_BROKER)
   dataKafkaProps.setProperty("group.id",
     ModelServingConfiguration.DATA_GROUP)
   dataKafkaProps.setProperty("auto.offset.reset", "latest")
   val modelKafkaProps = new Properties
   modelKafkaProps.setProperty("zookeeper.connect",
     ModelServingConfiguration.LOCAL_ZOOKEEPER_HOST)
   modelKafkaProps.setProperty("bootstrap.servers",
      ModelServingConfiguration.LOCAL_KAFKA_BROKER)
   modelKafkaProps.setProperty("group.id",
      ModelServingConfiguration.MODELS_GROUP)
   modelKafkaProps.setProperty("auto.offset.reset", "latest")
   // Create a Kafka consumer
   val dataConsumer = new FlinkKafkaConsumer010[Array[Byte]](...
   val modelConsumer = new FlinkKafkaConsumer010[Array[Byte]](...

   // Create input data streams
   val modelsStream = env.addSource(modelConsumer)
   val dataStream = env.addSource(dataConsumer)
   // Read data from streams
   val models = modelsStream.map(ModelToServe.fromByteArray(_))
     .flatMap(BadDataHandler[ModelToServe]).keyBy(_.dataType)
   val data = dataStream.map(DataRecord.fromByteArray(_))
     .flatMap(BadDataHandler[WineRecord]).keyBy(_.dataType)

   // Merge streams
   Data
.connect(models)
.process(DataProcessor())
 }
}
```

The workhorse of this implementation is the `buildGraph` method. It first configures and creates two Kafka consumers, for models and data, and then builds two input data streams from these consumers. It then reads data from both

streams and merges them.

The `FlinkKafkaConsumer010` class requires the definition of the deserialization schema. Because our messages are protobuf encoded, I treat Kafka messages as binary blobs. To do this, it is necessary to implement the `ByteArraySchema` class, as shown in Example 4-7, defining encoding and decoding of Kafka data.

*Example 4-7. The ByteArraySchema class*

```
class ByteArraySchema extends DeserializationSchema[Array[Byte]]
  with SerializationSchema[Array[Byte]] {

 override def isEndOfStream(nextElement:Array[Byte]):Boolean = false
 override def deserialize(message:Array[Byte]):Array[Byte] = message
 override def serialize(element: Array[Byte]): Array[Byte] = element
 override def getProducedType: TypeInformation[Array[Byte]] =
   TypeExtractor.getForClass(classOf[Array[Byte]])
}
```

## Using Partition-Based Joins

Flink's `RichCoFlatMapFunction` allows merging of two streams in parallel. A task is split into several parallel instances for execution with each instance processing a subset of the task's input data. The number of parallel instances of a task is called its parallelism.

When using this API on the partitioned stream, data from each partition is processed by a dedicated Flink executor. Records from the model stream are broadcast to all executors. As Figure 4-3 demonstrates, each partition of the input stream is routed to the corresponding instance of the model server. If the number of partitions of the input stream is less than Flink parallelism, only some of the model server instances will be utilized. Otherwise, some of the model server instances will serve more than one partition.

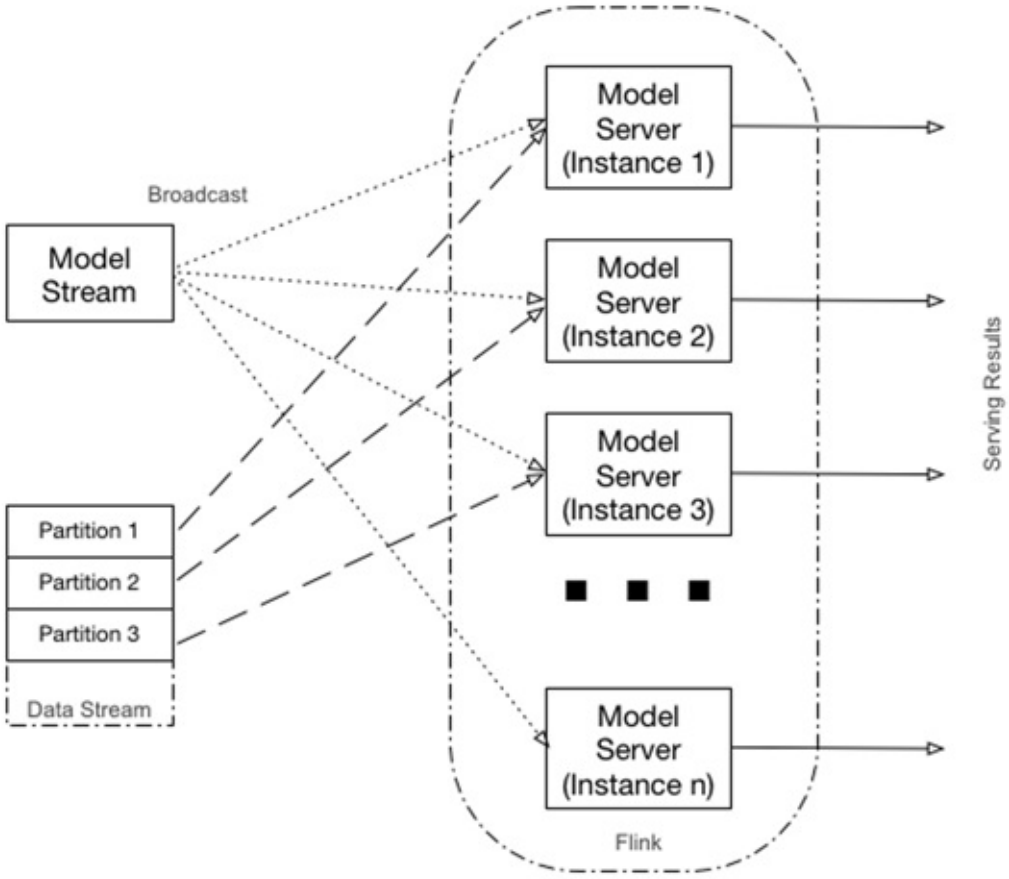



*Figure 4-3. Partition-based join*

Here are the main characteristics of this approach:

- The same model can be scored in one of several executors based on the partitioning of the data streams, which means that scaling of Flink (and input data partitioning) leads to better scoring throughput.
- Because the model stream is broadcast to all model server instances, which operate independently, some race conditions in the model update can exist, meaning that at the point of the model switch, some model jitter (models can be updated at different times in different instances, so for some short period of time different input records can be served by different models) can occur.

Based on these considerations, using global joins is an appropriate approach for the situations when it is necessary to score with one or a few models under heavy data load.

In the heart of this implementation is the `DataProcessorMap` class, which you can see in action in Example 4-8 (complete code available here).

*Example 4-8. The DataProcessMap class*

```
class DataProcessorMap
  extends RichCoFlatMapFunction[WineRecord, ModelToServe, Double]
  with CheckpointedFunction
  with CheckpointedRestoring[List[Option[Model]]] {
 ...

 override def flatMap2(model: ModelToServe, out: Collector[Double]):
   Unit = {
   import DataProcessorMap._
   println(s"New model - $model")
   newModel = factories.get(model.modelType) match{
     case Some(factory) => factory.create(model)
     case _ => None
   }
 }
 override def flatMap1(record: WineRecord, out: Collector[Double]):
   Unit = {
   // See if we need to update
   newModel match {
     case Some(model) => {
       // Close current model first
       currentModel match {
         case Some(m) => m.cleanup();
         case _ =>
       }
       // Update model
       currentModel = Some(model)
       newModel = None
     }
     case _ =>
   }
   currentModel match {
     case Some(model) => {
       val start = System.currentTimeMillis()
       val quality = model.score(record.asInstanceOf[AnyVal])
         .asInstanceOf[Double]
       val duration = System.currentTimeMillis() - start
     }
     case _ => println("No model available - skipping")
   }
 }
}
```

This implementation is very similar to the `DataProcessor` class (Example 4-1).

Following are the main differences between the two:

- The `DataProcessMap` class extends `RichCoFlatMapFunction`, whereas the `DataProcessor` class extends the `CoProcessFunction` class.
- The method names are different: `flatMap1` and `flatMap2` versus `processElement1` and `processElement2`. But the actual code within the methods is virtually identical.

Similar to the `DataProcessor` class, this class also implements support for checkpointing of state.

Overall orchestration of the execution is done using a Flink driver, which differs from the previous Flink driver for key-based joins (Example 4-6) only in how streams are delivered to the executors (`keyBy` versus `broadcast`) and processed (`process` versus `flatMap`) and joined, as shown in Example 4-9 (complete code available here).

*Example 4-9. Flink driver for global joins*

```
// Read data from streams
val models = modelsStream.map(ModelToServe.fromByteArray(_))
 .flatMap(BadDataHandler[ModelToServe]).broadcast
val data = dataStream.map(DataRecord.fromByteArray(_))
 .flatMap(BadDataHandler[WineRecord])
// Merge streams
Data
 .connect(models)
 .flatMap(DataProcessorMap())
```

Although this example uses a single model, you can easily expand it to support multiple models by using a map of models keyed on the data type.

A rich streaming semantic provided by Flink low-level process APIs provides a very powerful platform for manipulating data streams, including their transformation and merging. In this chapter, you have looked at different approaches for implementing proposed architecture using Flink. In Chapter 5, we look at how you can use Beam for solving the same problem.

# Chapter 5. Apache Beam Implementation

Beam is an open source, unified model for defining both batch and streaming processing pipelines. It is not a stream processing engine (SPE), but rather an SDK with which you can build a pipeline definition that can be executed by one of Beam's supported distributed processing backends. Using Beam, you can do the following:

- Use a single programming model for both batch and streaming use cases.
- Through separation of building and execution, you can use the same Beam pipelines on multiple execution environments.
- Write and share new SDKs, IO connectors, and transformation libraries, regardless of the specific runner.

Let's take a look at how you can use Beam's semantics to implement our solution.

## Overall Architecture

Beam provides very rich execution semantics for stream merging including `CoGroupByKey` and `Combine`. It also supports side inputs for bringing calculations on one stream as an input for processing in another stream. Unfortunately, all of these APIs are designed for windowed streams and do not work for the global windows—this is the problem that I am trying to solve.

The only option that I have found is using `Flatten`, which allows you to merge multiple streams into one. Combining this with the state feature used to store a model provides a reasonable approach for overall implementation, as shown in Figure 5-1.

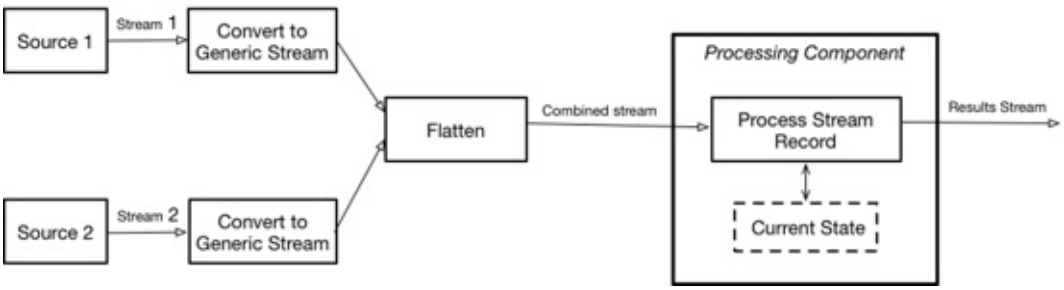



*Figure 5-1. The Beam implementation approach*

The caveat for using the `Flatten` operator is that all combined streams must have the same data definition. To satisfy this requirement, I have introduced the `DataWithModel` data structure (described momentarily), which can contain either data or a model definition.

## Implementing Model Serving Using Beam

Example 5-1 shows the overall pipeline for model serving using Beam (Beam provides only Java and Python APIs; as a result, code in this section is Java [complete code available here]):

*Example 5-1. Beam pipeline for model serving*

```
public class ModelServer {
   public static void main(String[] args) {
       // Create and initialize pipeline
       KafkaOptions options =
         JobConfiguration.initializePipeline(args);
       Pipeline p = Pipeline.create(options);

       // Coder to use for Kafka data - raw byte message
       KvCoder<byte[], byte[]> kafkaDataCoder =
       KvCoder.of(NullableCoder.of(ByteArrayCoder.of()),
         ByteArrayCoder.of());

       // Data Stream - gets data records from Kafka topic
       PCollection<KV<String, ModelServer1Support.DataWithModel>>
         dataStream = p
          .apply("data", KafkaIO.readBytes()
             .withBootstrapServers(options.getBroker())
             .withTopics(Arrays.asList(options.getKafkaDataTopic()))
             .updateConsumerProperties(
               JobConfiguration.getKafkaConsumerProps(options,
                 true))
             .withoutMetadata()).setCoder(kafkaDataCoder)
```

```
        // Convert Kafka message to WineRecord
        .apply("Parse data records", ParDo.of(
          new ModelServer1Support.ConvertDataRecordFunction()));

    // Models Stream - get model records from Kafka
    PCollection<KV<String,ModelServer1Support.DataWithModel>>
      modelStream = p
       .apply("models", KafkaIO.readBytes()
          .withBootstrapServers(options.getBroker())
          .withTopics(Arrays.asList(
            options.getKafkaModelsTopic()))
          .updateConsumerProperties(
            JobConfiguration.getKafkaConsumerProps(options,
              false))
          .withoutMetadata()).setCoder(kafkaDataCoder)
        // Convert Kafka record to ModelDescriptor
        .apply("Parse model", ParDo.of(
          new ModelServer1Support.ConvertModelRecordFunction()));
    // Create a combined PCollection stream.
    PCollection<KV<String,ModelServer1Support.DataWithModel>>
      combinedStream =
        PCollectionList.of(dataStream).and(modelStream)
                // Flatten the list
                .apply(Flatten.pCollections());

    // Score models and print result
    PCollection<Double> scoringResults = combinedStream
        // Score data using current model
        .apply("Scoring the model", ParDo.of(
           new ModelServer1Support.ScoringdFunction()))
        // Print scoring result
        .apply("Print result", MapElements.via(
           new ModelServer1Support.SimplePrinterFn<>()));

    // Run the pipeline
    p.run();
}
```

This code begins by creating the pipeline and setting up a Kafka coder. Because both streams contain only messages with no key, you must use `NullableCoder` for the key. Then, it defines two input streams, a data stream and a model stream, combining them together. Finally, the combined stream is used for model serving.

For reading from Kafka, I am using the new Beam support for Kafka, Kafka.IO, which reads byte arrays from Kafka and then applies transformation on this data to create a `PCollection` of `DataWithModel`, is definied in Example 5-2

(complete code available here).

*Example 5-2. The DataWithModel class*

```
public static class DataWithModel implements Serializable {
   private Winerecord.WineRecord data;
   private ModelDescriptor model;
   ...
   public Winerecord.WineRecord getData() {return data;}

   public ModelDescriptor getModel() {return model;}
}
```

Here `ModelDescriptor` is a Java version of the previous `ModelToServe` class (Example 4-2) and `WineRecord` is a representation of the data protobuf definition (Example 3-4).

Example 5-3 shows you how to handle the transformation of the data input to `DataWithModel` (complete code available here):

*Example 5-3. Converting data stream to DataWithModel*

```
public class ConvertDataRecordFunction
  extends DoFn<KV<byte[], byte[]>, KV<String, DataWithModel>> {
   @ProcessElement
   public void processElement(DoFn<KV<byte[], byte[]>,
      KV<String, DataWithModel>>.ProcessContext ctx) {

       // Get current element
       KV<byte[], byte[]> input = ctx.element();
       try {
           Winerecord.WineRecord record =
             Winerecord.WineRecord.parseFrom(input.getValue());
           ctx.output(KV.of(record.getDataType(),
             new DataWithModel(record)));
       } catch (Throwable t) {
          ...
       }
   }
}
```

Transformation of the model input to `DataWithModel` is equall simple.

The actual model serving uses state support and is implemented by the class shown in Example 5-4 (complete code available here):

*Example 5-4. Implementation of the model scoring*

```
public class ScoringFunction
```

```
extends DoFn<KV<String,DataWithModel>, Double> {

 private static final Map<Integer, ModelFactory> factories =
   new HashMap<Integer, ModelFactory>() {
 ...
 // Internal state
 @StateId("model")
 private final StateSpec<ValueState<Model>> modelSpec =
                   StateSpecs.value(ModelCoder.of());

 @ProcessElement
 public void processElement
   (DoFn<KV<String,DataWithModel>, Double>.ProcessContext ctx,
   @StateId("model") ValueState<Model> modelState) {
     // Get current element
     KV<String, DataWithModel> input = ctx.element();
     // Check if we got the model
     CurrentModelDescriptor descriptor =
       input.getValue().getModel();
     // Get current model
     Model model = modelState.read();
     if (descriptor != null) {
         // Process model - store it
         ModelFactory factory = factories
           .get(descriptor.getModelType().ordinal());
         if (factory == null)
             System.out.println("Unknown model type ");
         else {
             Optional<Model> current = factory.create(descriptor);
             if (current.isPresent()) {
                 if (model != null) model.cleanup();
                 // Create and store the model
                 modelState.write(current.get());
             } else
                 ...
         }
     }
     // Process data
     else {
         if (model == null)
             System.out.println("No model available - skipping");
         else {
             // Score the model
             long start = System.currentTimeMillis();
             double quality = (double) model.score(input
               .getValue().getData());
             long duration = System.currentTimeMillis() - start;
             // Propagate result
             ctx.output(quality);
```

```
            }
        }
    }
}
```

This class gets the next element from the stream and checks its type. If this is a model element, it updates the current model. If it is a data element, it is scored (assuming a model is present). This code is very similar to the `DataProcessor` class (Example 4-1) from the Flink implementation, with a significant difference: Flink's `DataProcessor` class provides two different methods that can be invoked on two different threads so that any extended processing time for model loading does not affect scoring significantly. In the case of Beam, it's a single method, so any delay in model creation can affect scoring.

Beam programs require a runner, and Flink is a popular choice. To be able to run this implementation on the Flink runner, which supports checkpointing, it is also necessary to implement a coder for the model object, as shown in Example 5-5 (complete code available here).

*Example 5-5. Model coder*

```
public class ModelCoder extends AtomicCoder<Model> {
...
 public static ModelCoder of() {
   return INSTANCE;
 }
...
 private static void writeModel(Model value, DataOutputStream dos)
 throws IOException {
   byte[] bytes = value.getBytes();
   VarInt.encode((long) bytes.length, dos);
   dos.write(bytes);
   VarInt.encode(value.getType(), dos);
 }

 private static Model readModel(DataInputStream dis)
 throws IOException {
   int len = (int)VarInt.decodeLong(dis);
   ...
   byte[] bytes = new byte[len];
   dis.readFully(bytes);
   int type = (int)VarInt.decodeLong(dis);
   ModelFactory factory = factories.get(type);
   ...
   return factory.restore(bytes);
 }
```

```
...
 @Override
 public void encode(Model value, OutputStream outStream,
      Context context) throws IOException {
 throws IOException {
   if (value == null)
     throw new CoderException("cannot encode a null model");
   if (context.isWholeStream) {
     byte[] bytes = value.getBytes();
     byte[] types = ByteUtils.longToBytes(value.getType());
     if (outStream instanceof ExposedByteArrayOutputStream) {
      ((ExposedByteArrayOutputStream) outStream).writeAndOwn(bytes);
     } else {
       outStream.write(bytes);
       outStream.write(types);
     }
   } else {
     writeModel(value, new DataOutputStream(outStream));
   }
 }
...
 @Override
 public Model decode(InputStream inStream, Context context)
   throws IOException {
   if (context.isWholeStream) {
     byte[] bytes = StreamUtils.getBytes(inStream);
     int type = (int)ByteUtils
       .bytesToLong(StreamUtils.getBytes(inStream));
     ModelFactory factory = factories.get(type);
     if (factory == null) {
       System.out.println("Unknown model type " + type);
       return null;
     }
     return factory.restore(bytes);
   } else {
     try {
       return readModel(new DataInputStream(inStream));
     } catch (EOFException | UTFDataFormatException exn) {
       ...
     }
   }
 }
...
}
```

Although this example uses a single model, you can expand it easily to support multiple models by using a state in the form of a map of models keyed on the data type.

Beam allows you to build an execution pipeline implementing model serving, which can be executed using multiple runners, including Apache Apex, Apache Flink, Apache Spark, and Google Cloud Dataflow. In Chapter 6, we look at how you can use Spark streaming to solve the same problem.

# Chapter 6. Apache Spark Implementation

Spark is a powerful open source processing engine built around speed, ease of use, and sophisticated analytics. It is currently the largest open source community in big data, with more than 1,000 contributors representing more than 250 organizations. The main characteristics of Spark are as follows:

Performance

: It is engineered from the bottom up for performance and can be very fast by exploiting in-memory computing and other optimizations.

Ease of use

: It provides easy-to-use APIs for operating on large datasets including a collection of more than 100 operators for transforming data and data frame APIs for manipulating semi-structured data.

Unified engine

: It is packaged with higher-level libraries, including support for SQL queries, streaming data, machine learning, and graph processing. These standard libraries increase developer productivity and can be seamlessly combined to create complex workflows.

Spark Streaming is an extension of core Spark API, which makes it easy to build fault-tolerant processing of real-time data streams. In the next section, we discuss how to use Spark Streaming for implementing our solution.

## Overall Architecture

Spark Streaming currently uses a minibatch approach to streaming, although a true streaming implementation is under way (discussed shortly). Because our problem requires state, the only option for our implementation is to use the `mapWithState` operator, where state is a Resilient Distributed Dataset core data

collection abstraction in Spark. Each record in this RDD is a key–value pair, in which the key is the data type (see Example 3-2) and the value is the current model object. This approach effectively uses the state RDD as a memory for the model, which is saved between minibatch executions.

The second issue for this implementation is the necessity to merge both input streams: the data and the model streams. Similar to the Beam case, the only option to merge nonwindowed streams in Spark is the `union` operator, which requires both streams to have the same data definition. Figure 6-1 presents the overall architecture for the Spark implementation.

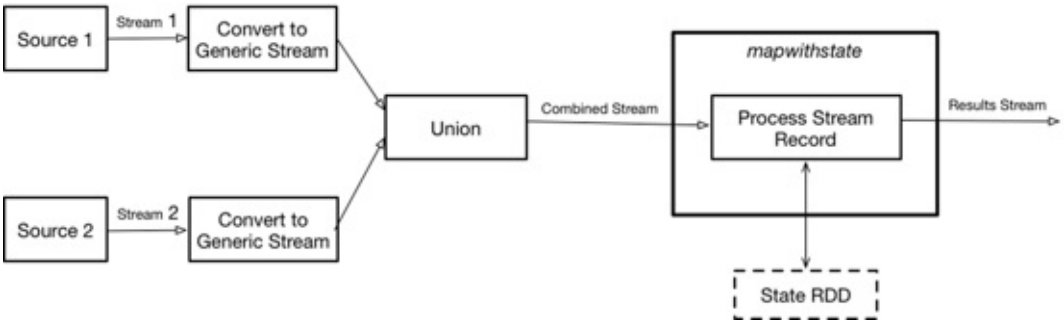



*Figure 6-1. Spark implementation approach*

# Implementing Model Serving Using Spark Streaming

Example 6-1 shows the overall pipeline implementation for model serving using Spark adhering to this architecture (complete code available here).

*Example 6-1. Model serving with Spark*

```
object SparkModelServer {
...
 def main(args: Array[String]): Unit = {
...
   // Initial state RDD for current models
   val modelsRDD = ssc.sparkContext.emptyRDD[(String, Model)]

   // Create models kafka stream
   val kafkaParams = KafkaSupport.getKafkaConsumerConfig(
     ModelServingConfiguration.LOCAL_KAFKA_BROKER)
   val modelsStream = KafkaUtils.createDirectStream[Array[Byte],
      Array[Byte]](ssc,PreferConsistent,Subscribe[Array[Byte],
         Array[Byte]]
```

```
(Set(ModelServingConfiguration.MODELS_TOPIC),kafkaParams))

// Create data kafka stream
val dataStream = KafkaUtils.createDirectStream[Array[Byte],
   Array[Byte]](ssc, PreferConsistent,Subscribe[Array[Byte],
     Array[Byte]]
(Set(ModelServingConfiguration.DATA_TOPIC),kafkaParams))

// Convert streams
val data = dataStream.map(r =>
DataRecord.fromByteArray(r.value())).filter(_.isSuccess)
 .map(d => DataWithModel(None, Some(d.get)))

val models = modelsStream.map(r =>
ModelToServe.fromByteArray(r.value())).filter(_.isSuccess)
   .map(m => DataWithModel(Some(m.get), None))

// Combine streams
val unionStream = ssc.union(Seq(data, models)).
  map(r => (r.getDataType, r))
// Score model using mapWithState
val mappingFunc = (dataType: String, dataWithModel:
    Option[DataWithModel], state: State[Model]) => {

  val currentModel = state.getOption().getOrElse(null
    .asInstanceOf[Model])
  dataWithModel match {
    case Some(value) =>
      if (value.isModel) {
        // Process model
        if (currentModel != null) currentModel.cleanup()
        val model = factories.get(value.getModel.modelType.value)
        match {
          case Some(factory) => factory.create(value.getModel)
          case _ => None
        }
        model match {
          case Some(m) => state.update(m)
          case _ =>
        }
        None
      }
      else {
        // Process data
        if (currentModel != null)
          Some(currentModel.
          score(value.getData.
            asInstanceOf[AnyVal]).asInstanceOf[Double])
        else
```

```
            None
          }
        case _ => None
      }
    }
    val resultDstream = unionStream.mapWithState(StateSpec.
      function(mappingFunc).initialState(modelsRDD))
    resultDstream.print()
    // Execute
    ssc.start()
    ssc.awaitTermination()
  }
}
```

The actual model serving is done in `mappingFunc` (its functionality is similar to the model serving implementation in Flink [Example 4-1]), which is invoked on the combined stream for every minibatch.

Another important point here is setting `KryoSerializer` as the default serializer and implementing Kryo serializers for the models. To register these serializers, you need to implement a special `KryoRegistrator` class, as shown in Example 6-2 (complete code available here).

*Example 6-2. Model Kryo serializer and registrator*

```
class ModelSerializerKryo extends Serializer[Model]{
...
 override def read(kryo: Kryo, input: Input, `type`: Class[Model]):
   Model = {
   import ModelSerializerKryo._

   val mType = input.readLong().asInstanceOf[Int]
   val bytes = Stream.continually(input.readByte()).
     takeWhile(_ != -1).toArray
   factories.get(mType) match {
     case Some(factory) => factory.restore(bytes)
     case _ => throw new Exception(s"Unknown model type $mType")
   }
 }

 override def write(kryo: Kryo, output: Output, value: Model):
   Unit = {
   println("KRYO serialization")
   output.writeLong(value.getType)
   output.write(value.toBytes)
 }
}
...
```

```
class ModelRegistrator extends KryoRegistrator {
 override def registerClasses(kryo: Kryo) {
   kryo.register(classOf[Model], new ModelSerializerKryo())
 }
}
```

Although this example uses a single model, you can expand it easily to support multiple models by using a map of models keyed on the data type.

When using this Spark implementation, it is necessary to keep in mind that it is not a true streaming system, meaning that there will be delay of up to the "batch time size," typically 500 milliseconds or more. If this is acceptable, this solution is very scalable.

Spark provides a new streaming API, called Structured Streaming, but you can't use it for the problem that we are solving here because some required operations are not yet supported (as of version 2.2); for example:

> *Any kind of joins between two streaming Datasets are not yet supported.*

Spark Streaming is the last stream-processing engine that I will discuss. In Chapters 7 and 8, we look at using streaming frameworks, starting with Kafka Streams, for solving the same problem.

# Chapter 7. Apache Kafka Streams Implementation

As described earlier, unlike Flink, Beam, and Spark, Kafka Streams is a client library for processing and analyzing data stored in Kafka. It builds upon important stream-processing concepts, such as properly distinguishing between event time and processing time, windowing support, and simple yet efficient management of application state. Because Kafka Streams is a Java Virtual Machine (JVM) framework, implementation of our architecture is fairly simple and straightforward. Because it runs in a single JVM, it is possible to implement state as a static Java object in memory, which can be shared by both streams. This in turn means that it is not necessary to merge streams—they can both execute independently while sharing the same state, which is the current model.

Although this solution will work, it has a serious drawback: unlike engines such as Spark or Flink, it does not provide automated recovery. Fortunately, Kafka Streams allows for implementation of state using custom state stores, which are backed up by a special Kafka topic. I will start this chapter by showing you how you can implement such a state store. Technically it is possible to use the key–value store provided by Kafka Streams, but I decided to create a custom state server to show implementation details.

## Implementing the Custom State Store

Implementing the custom state store begins with deciding on the data that it needs to hold. For our implementation, the only thing that needs to be stored is the current and new version of our models (complete code available here; currently Kafka Stream provides only Java APIs so as a result, all code in this section is Java):

*Example 7-1. StoreState class*

```
public class StoreState {
   private Model currentModel = null;
   private Model newModel = null;
```

```
...
   public Model getCurrentModel() {return currentModel;}
   public Model getNewModel() {return newModel;}
...
}
```

Kafka Streams supports several state store types including persistent and in-memory types that can further be set up as custom (user can define operations available on the store), key-value, or window stores (a window store differs from a key-value store in that it can store many values for any given key because the key can be present in multiple windows). For this application, shown in Example 7-2, I use an in-memory, custom store implementing the interface `org.apache.kafka.streams.processor.StateStore` (complete code available here).

*Example 7-2. ModelStateStore class*

```
public class ModelStateStore implements StateStore{
...
   @Override public void init(ProcessorContext context,
     StateStore root) {
       StateSerdes<Integer, StoreState> serdes =
                new StateSerdes<Integer, StoreState>(
                name, Serdes.Integer(), new ModelStateSerde());
       changeLogger = new ModelStateStoreChangeLogger(
         name, context, serdes);
       if (root != null && loggingEnabled) {
           context.register(root, loggingEnabled,
             new StateRestoreCallback() {
               @Override public void restore(byte[] key,
                 byte[] value) {
                   if (value == null) {
                       state.zero();
                   } else {
                       state = serdes.valueFrom(value);
                   }
               }
           });
       }
       open = true;
   }

   @Override public void flush() {
       if (loggingEnabled) {
           changeLogger.logChange(changelogKey, state);
       }
   }
```

```
...
}
```

This implementation uses `ModelStateStoreChangeLogger` for periodically writing the current state of the store to a special Kafka topic and to restore the store state on startup. Because `ModelStateStoreChangeLogger` uses a Kafka topic for storing the state, it requires a key. In our implementation, we can either use data type as a key or introduce a fake key to satisfy this requirement I used a fake key. It also uses `ModelStateSerde` for serializing/deserializing the content of the store to the binary blob, which can be stored in Kafka.

The two main methods in this class are `flush`, which is responsible for flushing the current state to Kafka, and `init`, which is responsible for restoring state from Kafka. Example 7-3 shows what the `ModelStateStoreChangeLogger` implementation looks like (complete code available here).

*Example 7-3. ModelStateStoreChangeLogger class*

```
public class ModelStateStoreChangeLogger<K,V> {
...
   public void logChange(K key, V value) {
       if (collector != null) {
           Serializer<K> keySerializer =
             serialization.keySerializer();
           Serializer<V> valueSerializer =
             serialization.valueSerializer();
           collector.send(this.topic, key, value, this.partition,
           context.timestamp(), keySerializer, valueSerializer);
       }
   }
}
```

This is a fairly simple class with one executable method, `logChange`, which sends the current state of the store to the topic assigned by the Kafka Streams runtime.

The second class used by `ModelStateStore` is `ModelStateSerde`, which you can see in Example 7-4 (complete code available here).

*Example 7-4. State serializer/deserializer class*

```
public class ModelStateSerde implements Serde<StoreState> {
   ...
   @Override public Serializer<StoreState> serializer() {
     return serializer;}
```

```
@Override public Deserializer<StoreState> deserializer() {
  return deserializer;}

public static class ModelStateSerializer implements Serializer<
  StoreState> {
    ...
    @Override public byte[] serialize(String topic, StoreState
      state) {

        bos.reset();
        DataOutputStream output = new DataOutputStream(bos);
        writeModel(state.getCurrentModel(), output);
        writeModel(state.getNewModel(), output);
        try {
            output.flush();
            output.close();
        }
        catch(Throwable t){}
        return bos.toByteArray();
    }
    private void writeModel(Model model,DataOutputStream output){
        try{
            if(model == null){
                output.writeLong(0);
                return;
            }
            byte[] bytes = model.getBytes();
            output.writeLong(bytes.length);
            output.writeLong(model.getType());
            output.write(bytes);
        }
        catch (Throwable t){
            ...
        }
    }
    ...
}
public static class ModelStateDeserializer
  implements Deserializer<StoreState> {
    ...
    @Override
    public StoreState deserialize(String topic, byte[] data) {

        ByteArrayInputStream bis =new ByteArrayInputStream(data);
        DataInputStream input = new DataInputStream(bis);
        Model currentModel = readModel(input);
        Model newModel = readModel(input);
        return new StoreState(currentModel, newModel);
```

```
        }

        @Override
        public void close() {}

        private Model readModel(DataInputStream input) {
            try {
                int length = (int)input.readLong();
                if (length == 0)
                    return null;
                int type = (int) input.readLong();
                byte[] bytes = new byte[length];
                input.read(bytes);
                ModelFactory factory = factories.get(type);
                return factory.restore(bytes);
            } catch (Throwable t) {
                ...
            }
        }
    }
}
```

This listing contains three classes:

- `ModelStateSerializer` is responsible for serialization of state to a byte array.
- `ModelStateDeserializer` is responsible for restoring of state from a byte array.
- `ModelStateSerde` is responsible for interfacing with the Kafka Streams runtime and providing classes for serialization/deserialization of state data.

The final class that is required for the model state store is `ModelStateStoreSupplier` to create store instances, which you can see in Example 7-5 (complete code available here).

*Example 7-5. ModelStateStoreSupplier class*

```
public class ModelStateStoreSupplier
  implements StateStoreSupplier<ModelStateStore> {
   ...
   @Override public ModelStateStore get() {
       return new ModelStateStore(name, loggingEnabled);
   }

   @Override public Map<String, String> logConfig() {
```

```
    return logConfig;}

  @Override public boolean loggingEnabled() {
    return loggingEnabled;}
}
```

# Implementing Model Serving

With the store in place, implementing model serving using Kafka Streams becomes very simple; it's basically two independent streams coordinated via a shared store (somewhat similar to a Flink [Figure 4-1], with the state being implemented as a state store).

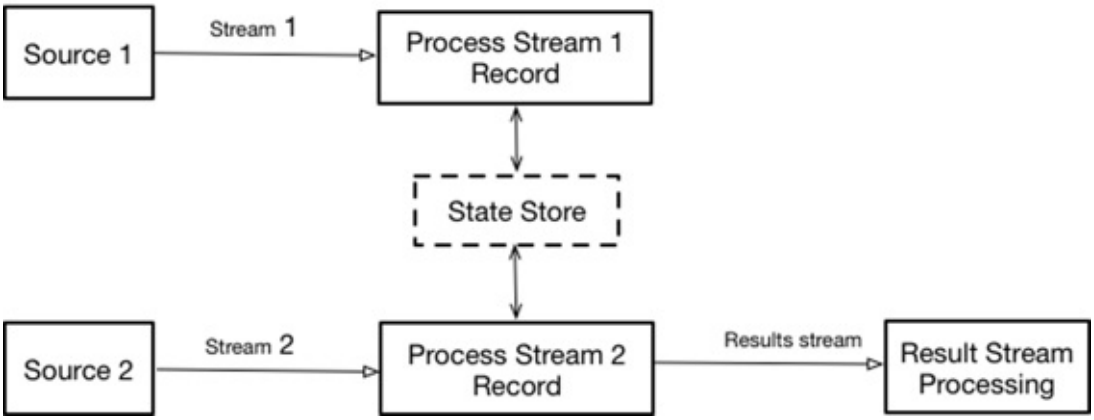



*Figure 7-1. Kafka Streams implementation approach*

Example 7-6 presents the overall implementation of this architecture (complete code available here).

*Example 7-6. Implementation of model serving*

```
public class ModelServerWithStore {
   public static void main(String [ ] args) throws Throwable {
       Properties streamsConfiguration = new Properties();
       // Give the Streams application a unique name.
       streamsConfiguration.put(
             StreamsConfig.APPLICATION_ID_CONFIG, "model-serving");
       streamsConfiguration.put(
             StreamsConfig.CLIENT_ID_CONFIG,"model_server-client");
       // Where to find Kafka broker(s).
       streamsConfiguration.put(
             StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
             ApplicationKafkaParameters.LOCAL_KAFKA_BROKER);
       final File example = Files.createTempDirectory(
             new File("/tmp").toPath(), "example").toFile();
```

```
        streamsConfiguration.put(StreamsConfig.STATE_DIR_CONFIG,
          example.getPath());
        // Create topology
        final KafkaStreams streams =
          createStreams(streamsConfiguration);
        streams.cleanUp();
        streams.start();
        // Add shutdown hook to respond to SIGTERM
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            try {
                streams.close();
            } catch (Exception e) {}
        }));
    }

    static KafkaStreams createStreams(
      final Properties streamsConfiguration) {

        Serde<StoreState> stateSerde = new ModelStateSerde();
        ByteArrayDeserializer deserializer =
          new ByteArrayDeserializer();
        ModelStateStoreSupplier storeSupplier =
               new ModelStateStoreSupplier("modelStore", stateSerde);
        KStreamBuilder builder = new KStreamBuilder();
        // Data input streams
        builder.addSource("data-source", deserializer, deserializer,
               ApplicationKafkaParameters.DATA_TOPIC)
               .addProcessor("ProcessData",
               DataProcessorWithStore::new,"data-source");
        builder.addSource("model-source", deserializer, deserializer,
               ApplicationKafkaParameters.MODELS_TOPIC)
               .addProcessor("ProcessModels",
               ModelProcessorWithStore::new,"model-source");
        builder.addStateStore(storeSupplier, "ProcessData",
               "ProcessModels");
        return new KafkaStreams(builder, streamsConfiguration);
    }
}
```

The main method of this class configures Kafka Streams, builds the streams topology, and then runs them. The execution topology configuration is implemented in the `CreateStreams` method. It first creates the store provider. Next it creates two effectively independent stream-processing pipelines. Finally, it creates a state store and attaches it to both pipelines (processors).

Each pipeline reads data from its dedicated stream and then invokes a dedicated processor.

Example 7-7 shows what the model processor looks like (complete code available here).

*Example 7-7. Model processor*

```
public class ModelProcessorWithStore
  extends AbstractProcessor<byte[], byte[]> {
...
   @Override
   public void process(byte[] key, byte[] value) {

       Optional<CurrentModelDescriptor> descriptor =
         DataConverter.convertModel(value);
       if(!descriptor.isPresent()){
           return;
       }
       CurrentModelDescriptor model = descriptor.get();
       System.out.println("New scoring model " + model);
       if(model.getModelData() == null) {
           System.out.println("Not yet supported");
           return;
       }
       ModelFactory factory = factories.get(model.getModelType()
         .ordinal());
       if(factory == null){
           System.out.println("Bad model type " +
             model.getModelType());
           return;
       }
       Optional<Model> current = factory.create(model);
       if(current.isPresent()) {
           modelStore.setNewModel(current.get());
           return;
       }
   }

   @Override
   public void init(ProcessorContext context) {
       this.context = context;
       this.context.schedule(10000);
       modelStore = (ModelStateStore)
         this.context.getStateStore("modelStore");
   }
}
```

This class has two important methods: `init`, which is responsible for connecting to the data store, and `process`, which is responsible for the actual execution of the processor's functionality. The process method unmarshals the incoming

model message, converts it to the internal representation, and then deposits it (as a new model) to the data store.

The data stream processor is equally simple, as demonstrated in Example 7-8 (complete code available here).

*Example 7-8. Data processor*

```
public class DataProcessorWithStore
  extends AbstractProcessor<byte[], byte[]> {
...
   @Override
   public void process(byte[] key, byte[] value) {
       Optional<Winerecord.WineRecord> dataRecord =
         DataConverter.convertData(value);
       if(!dataRecord.isPresent()) {
           return;                                  // Bad record
       }
       if(modelStore.getNewModel() != null){
           // update the model
           if(modelStore.getCurrentModel() != null)
               modelStore.getCurrentModel().cleanup();
           modelStore.setCurrentModel(modelStore.getNewModel());
           modelStore.setNewModel(null);
       }
       // Actually score
       if(modelStore.getCurrentModel() == null) {
           // No model currently
           System.out.println("No model available - skipping");
       }
       else{
           // Score the model
           double quality = (double) modelStore.getCurrentModel()
             .score(dataRecord.get());
           System.out.println(
               "Calculated quality - " + quality + " in " +
                 duration + "ms");
        }
   }

   @Override
   public void init(ProcessorContext context) {
       this.context = context;
       this.context.schedule(10000);
       modelStore = (ModelStateStore)
         this.context.getStateStore("modelStore");
       Objects.requireNonNull(modelStore, "Store can't be null");
   }
}
```

Similar to the model processor, this class has two methods with the same responsibilities. The `process` method unmarshals an incoming data message and converts it to the internal representation. It then checks whether there is a new model and updates the content of the state store if necessary. Finally, if the model is present, it scores input data and prints out the result.

Although the implementation presented here scores a single model, you can expand it easily to support multiple models using the data type for routing, as needed.

## Scaling the Kafka Streams Implementation

With all the simplicity of this implementation, the question remains about the scalability of this solution, given that the Kafka Streams implementation runs within a single Java Virtual Machine (JVM). Fortunately, the Kafka Streams implementation is easily scalable based on Kafka data partitioning. In effect Kafka Streams scaling is very similar to Flink's partition-based join approach (Figure 4-3). Keep in mind that for this to work, every instance must have a different consumer group for the model topic (to ensure that all instances are reading the same sets of models) and a same consumer group for the data topic (to ensure that every instance gets only part of data messages). Figure 7-2 illustrates Kafka Streams scaling.

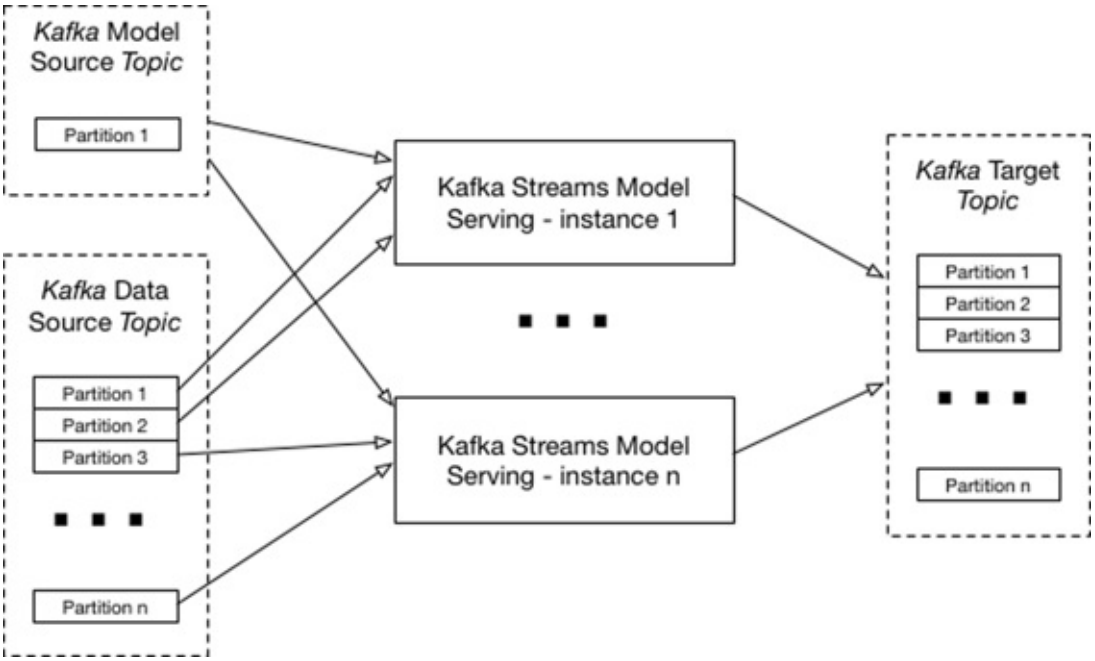


*Figure 7-2. Scaling Kafka Streams implementation*

Similar to Flink's partition-based joins implementation, a multi-JVM implementation of a Kafka Streams–based model server is prone to potential race conditions, as it does not guarantee that models are updated in all JVMs at the same time.

In Chapter 8, we will see how we can use another popular streaming framework—Akka Streams—to build model serving solutions.

# Chapter 8. Akka Streams Implementation

Akka Streams, part of the Akka project, is a library focused on in-process back-pressured reactive streaming. It is similar to Kafka Streams, but it is not strictly bound to Kafka; it provides a broad ecosystem of connectors to various technologies (datastores, message queues, file stores, streaming services, etc).

This connectors ecosystem is collectively referred to as the Alpakka ecosystem and builds on the Reactive Streams standard (which Akka Streams has been coleading since its inception). And with Reactive Streams having become part of Java in version 9, via the JEP-266 (More Concurrency Updates) process, even more connectors and exponential growth in this connectors ecosystem are bound to happen, with major players like Amazon already having adopted it in the new AWS 2.0 SDK.

In Akka Streams computations are written in graph-resembling domain-specific lanauge (DSL), which aims to make translating graph drawings to and from code simpler.

Graphs in Akka Streams are built from the following base elements:

- Source is a partial graph with exactly one output.
- Sink is a partial graph with exactly one input.
- Stage is an implementation of a transformation from input(s) to output(s).
- Flow is a partial graph (stage) with exactly one input and exactly one output.
- Fan-in is a partial graph (stage) that takes multiple streams as an input and provides a single output combining the elements from all of the inputs in user-defined ways.
- Fan-out is a partial graph (stage) that takes one input and produces multiple outputs. They might route the elements between different outputs, or emit

elements on multiple outputs at the same time.

- Custom stage is a partial graph can take an arbitrary number of inputs, apply custom logic, and produce an arbitrary number of outputs.

Out of the box, Akka Streams provides quite a few built-in stages that you can use for building custom applications.

## Overall Architecture

For model serving implementation, I decided to use a custom stage, which is a fully type-safe way to encapsulate required functionality. Our stage will provide functionality somewhat similar to a Flink low-level join (Figure 4-1).

With such a component in place, the overall implementation is going to look that shown in Figure 8-1.

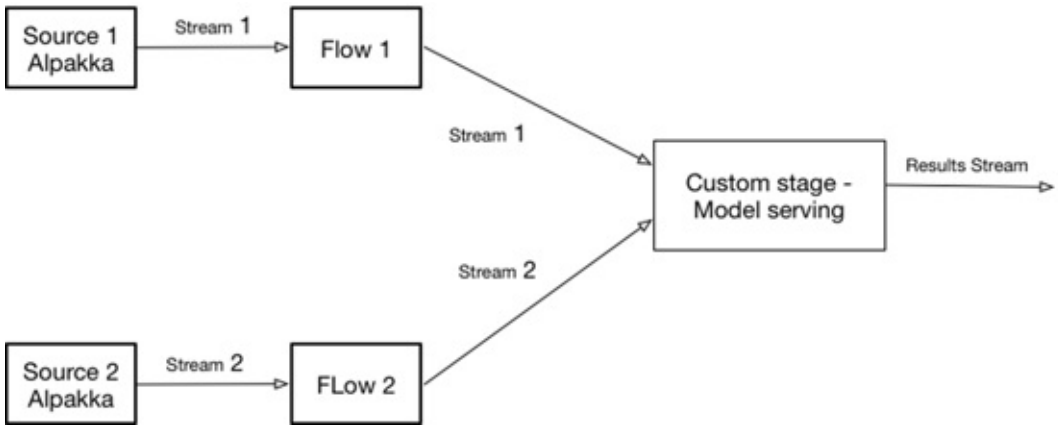



*Figure 8-1. Akka Streams implementation approach*

## Implementing Model Serving Using Akka Streams

Implementing a custom `GraphStage` is an advanced topic in Akka, so you might want to check the documentation. Implementation begins by defining the stage's shape, as shown in Example 8-1 (complete code available here).

*Example 8-1. Defining a stage's shape*

```
class ModelStageShape() extends Shape {
```

```
 var dataRecordIn = Inlet[WineRecord]("dataRecordIn")
 var modelRecordIn = Inlet[ModelToServe]("modelRecordIn")
 var scoringResultOut = Outlet[Option[Double]]("scoringOut")

def this(dataRecordIn: Inlet[WineRecord], modelRecordIn:
  Inlet[ModelToServe],scoringResultOut: Outlet[Option[Double]]) {

   this()
   this.dataRecordIn = dataRecordIn
   this.modelRecordIn = modelRecordIn
   this.scoringResultOut = scoringResultOut
 }

 override def deepCopy(): Shape =
   new ModelFanInShape(dataRecordIn.carbonCopy(),
   modelRecordIn.carbonCopy(), scoringResultOut)

 override def copyFromPorts(inlets: immutable.Seq[Inlet[_]],
   outlets: immutable.Seq[Outlet[_]]): Shape =
   new ModelFanInShape(
     inlets(0).asInstanceOf[Inlet[WineRecord]],
     inlets(1).asInstanceOf[Inlet[ModelToServe]],
     outlets(0).asInstanceOf[Outlet[Option[Double]]])

 override val inlets = List(dataRecordIn, modelRecordIn)
 override val outlets = List(scoringResultOut)
}
```

Example 8-1 defines a shape with two inputs, data records and model records, and one output, the scoring results. Shape also defines the types associated with the inputs and output (Akka Streams is strongly typed).

With this in place, the stage implementation is presented in Example 8-2 (complete code available here).

*Example 8-2. Stage implementation*

```
class ModelStage
  extends GraphStageWithMaterializedValue[ModelStageShape] {

 private val factories = ...

 override val shape: ModelStageShape = new ModelStageShape

 override def createLogicAndMaterializedValue
   (inheritedAttributes: Attributes): (GraphStageLogic) = {

   new GraphStageLogicWithLogging(shape) {
     // State must be kept in the Logic instance
```

```
private var currentModel : Option[Model] = None
private var newModel : Option[Model] = None
// Start pulling input streams
override def preStart(): Unit = {
  tryPull(shape.modelRecordIn)
  tryPull(shape.dataRecordIn)
}

setHandler(shape.modelRecordIn, new InHandler {
  override def onPush(): Unit = {
    val model = grab(shape.modelRecordIn)
    newModel = factories.get(model.modelType) match{
      case Some(factory) => factory.create(model)
      case _ => None
    }
    pull(shape.modelRecordIn)
  }
})

setHandler(shape.dataRecordIn, new InHandler {
  override def onPush(): Unit = {
    val record = grab(shape.dataRecordIn)
    newModel match {
      case Some(model) => {
        // Close current model first
        currentModel match {
          case Some(m) => m.cleanup()
          case _ =>
        }
        // Update model
        currentModel = Some(model)
        newModel = None
      }
      case _ =>
    }
    currentModel match {
      case Some(model) => {
        val quality =
         model.score(record.asInstanceOf[AnyVal])
           .asInstanceOf[Double]
        push(shape.scoringResultOut, Some(quality))
      }
      case _ => {
        push(shape.scoringResultOut, None)
      }
    }
    pull(shape.dataRecordIn)
  }
})
```

```
    setHandler(shape.scoringResultOut, new OutHandler {
      override def onPull(): Unit = {}
    })
  }
 }
}
```

The bulk of the implementation is the `createLogic` method, which defines the main processing logic of the stage. This is done by defining handlers for inputs and outputs.

The handler for model input creates a new model and stores it in a local variable for further usage. Because the handler supports backpressure, it controls the rate of records coming from the source (it ensures that processing of the current message is complete) before it tries to read the next record. This is achieved by explicitly polling for a new record at the end of the handler execution.

The handler for the data input checks whether there is a new model and, if so, it updates the model that it is serving. Then, it checks whether it has a model and, if so, it scores the data. Similar to the model handler, after the record processing is complete, the handler polls for the next record.

Finally, the output handler does nothing.

The other important method in this class is `preStart`, which initiates polling for data and model records.

With the stage implementation in place, the implementation of the server looks like Example 8-3 (complete code available here).

*Example 8-3. Akka model server implementation*

```
object AkkaModelServer {

 implicit val system = ActorSystem("ModelServing")
 implicit val materializer = ActorMaterializer()
 implicit val executionContext = system.dispatcher

 val dataConsumerSettings = ConsumerSettings(system,
  new ByteArrayDeserializer, new ByteArrayDeserializer)
   .withBootstrapServers(LOCAL_KAFKA_BROKER)
   .withGroupId(DATA_GROUP)
   .withProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest")

 val modelConsumerSettings = ConsumerSettings(
```

```
  system, new ByteArrayDeserializer, new ByteArrayDeserializer)
   .withBootstrapServers(LOCAL_KAFKA_BROKER)
   .withGroupId(MODELS_GROUP)
   .withProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest")

 def main(args: Array[String]): Unit = {
   import ApplicationKafkaParameters._
   val modelStream: Source[ModelToServe, Consumer.Control] =Consumer
       .atMostOnceSource(modelConsumerSettings,Subscriptions
       .topics(MODELS_TOPIC))
       .map(record => ModelToServe.fromByteArray(record.value()))
       .filter(_.isSuccess).map(_.get)

   val dataStream: Source[WineRecord, Consumer.Control] = Consumer
       .atMostOnceSource(dataConsumerSettings, Subscriptions
       .topics(DATA_TOPIC))
       .map(record => DataRecord.fromByteArray(record.value()))
       .filter(_.isSuccess).map(_.get)
   val model = new ModelStage()

   def dropMaterializedValue[M1, M2, M3](m1: M1, m2: M2, m3: M3):
     NotUsed = NotUsed

   val modelPredictions  = Source.fromGraph(
     GraphDSL.create(dataStream, modelStream, model)(
      dropMaterializedValue) {
       implicit builder => (d, m, w) =>
         import GraphDSL.Implicits._
         // Wire input streams with the model stage (2 in, 1 out)
         d ~> w.dataRecordIn
         m ~> w.modelRecordIn
         SourceShape(w.scoringResultOut)
     }
   )
}
```

Here, `ActorSystem` and `Materializer` are created, which is necessary for running any Akka Stream application. After that both data and model streams are created, each reading from a corresponding Kafka topic and transforming data from binary format to the internal representation. Finally, both streams are connected to our stage and the resulting graph is started.

## Scaling Akka Streams Implementation

Similar to Kafka Streams, the Akka Streams implementation runs within a single

Java Virtual Machine (JVM). Scaling this solution might require running multiple instances of our implementation on multiple machines (similar to Kafka Streams application scaling [see Figure 7-2]). Because the implementation is using specific consumer groups for reading data records, Kafka will realize that they belong to the same application and send different partitions to different instances, thus implementing partition-based load balancing.

## Saving Execution State

An additional deficiency of our simple implementation is the fact that our implementation of the stage is not persistent, which means that a crash could lose state. Akka Streams by itself does not provide a persistency model. Akka Persistence, a separate module of Akka, does provide an event source–based library for persistence. Akka Streams provides support for usage of stage actors, where a stage can encapsulate a custom actor. This allows usage of persistent actors, Akka's standard way of persisting state in the system, thus solving persistence problems for such applications.

In Chapters 4 through 8, you have seen how you can implement model serving by using different streaming engines and frameworks, but as with any streaming application it is necessary to provide a well-defined monitoring solution. An example of information that you might want to see for a model serving application includes:

- Which model is currently used?
- When was it installed?
- How many times have models been served?
- What are the average and minimum/maximum model serving times?

We look at such solutions in Chapter 9.

# Chapter 9. Monitoring

General approaches for obtaining execution information include the following:

- Logging
- Writing this information (periodically) to an external storage (for example databases)

Although both of these approaches work and are often used, they typically suffer from the following:

- They introduce additional latency to the execution—they are typically implemented using synchronous calls within the execution itself.
- They require additional software components—database, log aggregators, and so on—which translates to additional maintenance

Both Flink and Kafka Streams recently introduced queryable state (Figure 9-1), which is a different approach to such monitoring.

The Kafka Streams documentation defines queryable state (interactive queries) as an approach, which, according to the documentation

> *lets you get more from streaming than just the processing of data. This feature allows you to treat the stream processing layer as a lightweight embedded database and, more concretely, to directly query the latest state of your stream processing application, without needing to materialize that state to external databases or external storage first.*

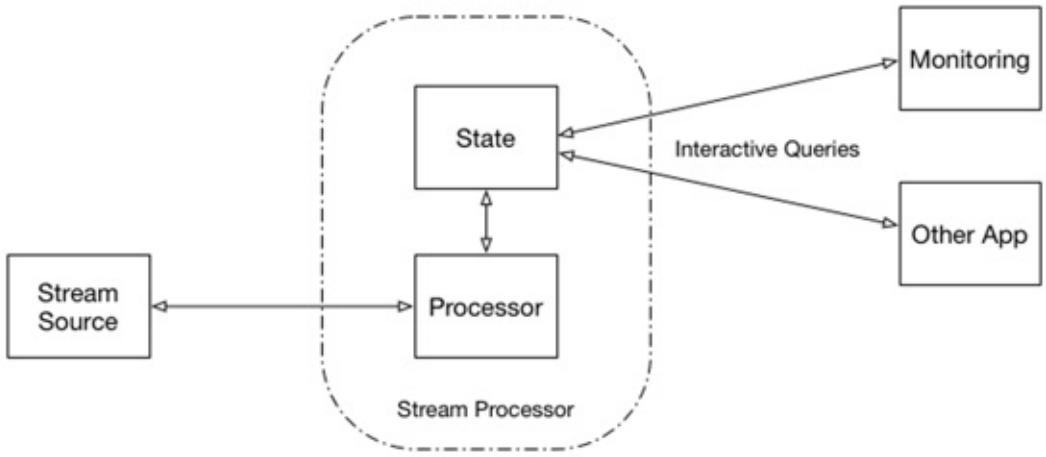



*Figure 9-1. Queryable state*

# Flink

Flink recently introduced a managed keyed-state interface that provides access to a state that is scoped to the key of the current execution. This means that this type of state can be used only on a Flink `KeyedStream`, and is applicable only to Flink's key-based joins implementation.

The current Flink implementation includes the following state options:

`ValueState<T>`

: This keeps a value that can be updated and retrieved. You can set the value using `update(T)` and retrieve it using `T value()`.

`ListState<T>`

: This keeps a list of elements. `ListState` supports appending elements and retrieving an `Iterable` over all currently stored elements. You add elements using `add(T)`, and retrieve an `Iterable` by using `Iterable<T> get()`.

`ReducingState<T>`

: This keeps a single value representing an aggregation of all values added to the state. Similar to `ListState`, you can add an element using `add(T)`, but the elements are reduced to an aggregate using a specified `ReduceFunction`.

`FoldingState<T, ACC>`

This keeps a single value that represents the aggregation of all values added to the state. Unlike `ReducingState`, the type of the aggregate can be different from the type of elements that are added to the state. Similar to `ReducingState`, you add elements by using `add(T)` and you fold them into an aggregate using a specified `FoldFunction`.

Flink also provides the `QueryableStateClient` class that you can use for queries against the KvState instances that serve the state internally. You need to configure it with a valid Flink `JobManager` address, port, and job ID.

To incorporate queryable statistics in our implementation, you must modify the `DataProcessor` class (Example 4-1) by adding in `ValueState<T>`. The updated version of this class looks like Example 9-1 (complete code available here).

*Example 9-1. The ModelToServeStats class*

```
class DataProcessorKeyed
    extends CoProcessFunction[WineRecord, ModelToServe, Double]
      with CheckpointedFunction
      with CheckpointedRestoring[List[Option[Model]]] {


 var modelState: ValueState[ModelToServeStats] = _
 var newModelState: ValueState[ModelToServeStats] = _
 ...

 override def open(parameters: Configuration): Unit = {
 val modelDesc = new ValueStateDescriptor[ModelToServeStats](
   "currentModel",
   createTypeInformation[ModelToServeStats])
   modelDesc.setQueryable("currentModel")

 modelState = getRuntimeContext.getState(modelDesc)
 val newModelDesc = new ValueStateDescriptor[ModelToServeStats](
   "newModel",
   createTypeInformation[ModelToServeStats])
 newModelState = getRuntimeContext.getState(newModelDesc)
}

override def processElement2(model: ModelToServe,
    ctx: CoProcessFunction[WineRecord,
    ModelToServe, Double]#Context, out: Collector[Double]): Unit = {


   import DataProcessorKeyed._
```

```
    println(s"New model - $model")
    newModelState.update(new ModelToServeStats(model))
    newModel = factories.get(model.modelType) match {
      case Some(factory) => factory.create (model)
      case _ => None
    }
  }

  override def processElement1(record: WineRecord,
     ctx: CoProcessFunction[WineRecord,
     ModelToServe, Double]#Context, out: Collector[Double]): Unit = {


     ...
     val start = System.currentTimeMillis()
     val quality = model.score(record
       .asInstanceOf[AnyVal]).asInstanceOf[Double]
     val duration = System.currentTimeMillis() - start
     modelState.update(modelState.value().incrementUsage(duration))
     ...
}
```

This addition tracks the current model name, the time it was introduced, number of usages, overall execution time, and minimum/maximum execution time.

You can access this information by using a queryable state client that you can implement as shown in Example 9-2 (complete code available here).

*Example 9-2. Flink queryable state client*

```
object ModelStateQuery {
  def main(args: Array[String]) {
    val jobId = JobID.fromHexString("...")
    val types = Array("wine")
    val config = new Configuration()
    config.setString(JobManagerOptions.ADDRESS, "localhost")
    config.setInteger(JobManagerOptions.PORT, 6124)
    ...
    val client = new QueryableStateClient(config,
      highAvailabilityServices)
    val execConfig = new ExecutionConfig
    val keySerializer = createTypeInformation[
      String].createSerializer(execConfig)
    val valueSerializer = createTypeInformation[ModelToServeStats]
      .createSerializer(execConfig)
    while(true) {
      val stats = for (key <- types) yield {
        val serializedKey = KvStateRequestSerializer
           .serializeKeyAndNamespace(key,keySerializer,
```

```
          VoidNamespace.INSTANCE, VoidNamespaceSerializer.INSTANCE)
        // now wait for the result and return it
        try {
          val serializedResult = client.getKvState(jobId,
            "currentModel",key.hashCode(), serializedKey)
          val serializedValue = Await.result(serializedResult,
            FiniteDuration(2, TimeUnit.SECONDS))
          val value = KvStateRequestSerializer.deserializeValue
            (serializedValue, valueSerializer)
          List(value.name, value.description, value.since,
            value.usage,value.duration, value.min, value.max)
    ...
     }
     stats.toList.filter(_.nonEmpty).foreach(row =>
...
}
```

This simple implementation polls the running Flink server every `timeInterval` and prints results. `jobId` here is a current `jobId` executed by a Flink server.

## Kafka Streams

When dealing with Kafka Streams, you must consider two things: what is happening in a single Java Virtual Machine (JVM), and how several JVMs representing a single Kafka Streams application work together. As a result, Kafka queryable APIs comprise two parts:

Querying local state stores (for an application instance)

> This provides access to the state that is managed locally by an instance of your application (partial state). In this case, an application instance can directly query its own local state stores. You can thus use the corresponding (local) data in other parts of your application code that are not related to calling the Kafka Streams API.

Querying remote state stores (for the entire application)

> To query the full state of an application, it is necessary to bring together local fragments of the state from every instance. This means that in addition to being able to query local state stores, it is also necessary to be able to discover all the running instances of an application. Collectively, these building blocks enable intra-application communications (between instances

of the same app) as well as inter-application communication (from other applications) for interactive queries.

Implementation begins with defining the state representation, `ModelServingInfo`, which can be queried to get information about the current state of processing, as demonstrated in Example 9-3 (complete code available here).

*Example 9-3. The ModelServingInfo class*

```
public class ModelServingInfo {

   private String name;
   private String description;
   private long since;
   private long invocations;
   private double duration;
   private long min;
   private long max;
   ...
   public void update(long execution){
       invocations++;
       duration += execution;
       if(execution < min) min = execution;
       if(execution > max) max = execution;
   }
...
}
```

Now, you must add this information to the state store shown in Example 7-2. Example 9-4 shows you how (complete code available here).

*Example 9-4. Updated StoreState class*

```
public class StoreState {
   ...
   private ModelServingInfo currentServingInfo = null;
   private ModelServingInfo newServingInfo = null;
   ...
   public ModelServingInfo getCurrentServingInfo() {
     return currentServingInfo;}

   public void setCurrentServingInfo(ModelServingInfo
     currentServingInfo) {
       this.currentServingInfo = currentServingInfo;
   }

   public ModelServingInfo getNewServingInfo() {
```

```
     return newServingInfo;}

   public void setNewServingInfo(ModelServingInfo newServingInfo) {
       this.newServingInfo = newServingInfo;
   }
}
```

This adds two instances of the `ModelServingInfo` class (similar to the `Model` class). Adding those will in turn require a change in `ModelSerde` (Example 7-4) to implement serialization/deserialization support for the `ModelServingInfo` class (complete code available here). Example 9-5 presents the code to do this.

*Example 9-5. ModelServingInfo serialization/deserialization*

```
...
private void writeServingInfo(ModelServingInfo servingInfo,
  DataOutputStream output){
   try{
       if(servingInfo == null) {
           output.writeLong(0);
           return;
       }
       output.writeLong(5);
       output.writeUTF(servingInfo.getDescription());
       output.writeUTF(servingInfo.getName());
       output.writeDouble(servingInfo.getDuration());
       output.writeLong(servingInfo.getInvocations());
       output.writeLong(servingInfo.getMax());
       output.writeLong(servingInfo.getMin());
       output.writeLong(servingInfo.getSince());
   }
   catch (Throwable t){
       System.out.println("Error Serializing servingInfo");
       t.printStackTrace();
   }
}
...
private ModelServingInfo readServingInfo(DataInputStream input) {
   try {
       long length = input.readLong();
       if (length == 0) return null;
       String descriprtion = input.readUTF();
       String name = input.readUTF();
       double duration = input.readDouble();
       long invocations = input.readLong();
       long max  = input.readLong();
       long min = input.readLong();
       long since = input.readLong();
       return new ModelServingInfo(name, descriprtion, since,
```

```
          invocations,duration, min, max); duration, min, max);
   } catch (Throwable t) {
       System.out.println("Error Deserializing serving info");
       t.printStackTrace();
       return null;
   }
}
```

Finally, you also must change the `ModelStateStore` class (Example 7-2). First, the Streams queryable state allows only read access to the store data, which requires introduction of an interface supporting only read access that can be used for query (Example 9-6):

*Example 9-6. Queryable state store interface*

```
public interface ReadableModelStateStore {
   ModelServingInfo getCurrentServingInfo();
}
```

With this in place, you can extend the `DataProcessorWithStore` class (Example 7-7) to collect your execution information, as shown in Example 9-7 (complete code available here).

*Example 9-7. Updated data processor class*

```
...
   // Score the model
   long start = System.currentTimeMillis();
   double quality = (double) modelStore.getCurrentModel().score(
     dataRecord.get());
   long duration = System.currentTimeMillis() - start;
   modelStore.getCurrentServingInfo().update(duration);
...
```

To make the full state of the application (all instances) queryable, it is necessary to provide discoverability of the additional instances. Example 9-8 presents a simple implementation of such a service (complete code available here).

*Example 9-8. Simple metadata service*

```
public class MetadataService {
   private final KafkaStreams streams;

   public MetadataService(final KafkaStreams streams) {
       this.streams = streams;
   }

   public List<HostStoreInfo> streamsMetadata() {
```

```
        // Get metadata for all of the instances of this application
        final Collection<StreamsMetadata> metadata =
          streams.allMetadata();
        return mapInstancesToHostStoreInfo(metadata);
    }

    public List<HostStoreInfo> streamsMetadataForStore(
      final  String store) {
        // Get metadata for all of the instances of this application
        final Collection<StreamsMetadata> metadata =
          streams.allMetadataForStore(store);
        return mapInstancesToHostStoreInfo(metadata);
    }

    private List<HostStoreInfo> mapInstancesToHostStoreInfo(
      final Collection<StreamsMetadata> metadatas) {
        return metadatas.stream().map(metadata -> new HostStoreInfo(
          metadata.host(),metadata.port(),
          metadata.stateStoreNames()))
          .collect(Collectors.toList());
    }
}
```

To actually be able to serve this information, you implement a simple REST service exposing information from the metadata service using an HTTP server implementation as well as a framework for building a REST service. In this example, I used Jetty and JAX-RS (with corresponding JAX-RS annotations), which are popular choices in the Java ecosystem. Example 9-9 shows a simple REST service implementation that uses metadata service and model serving information (complete code available here).

*Example 9-9. Simple REST service implementation*

```
@Path("state")
public class QueriesRestService {
    private final KafkaStreams streams;
    private final MetadataService metadataService;
    private Server jettyServer;

    public QueriesRestService(final KafkaStreams streams) {
        this.streams = streams;
        this.metadataService = new MetadataService(streams);
    }

    @GET()
    @Path("/instances")
    @Produces(MediaType.APPLICATION_JSON)
    public List<HostStoreInfo> streamsMetadata() {
```

```
        return metadataService.streamsMetadata();
    }
    @GET()
    @Path("/instances/{storeName}")
    @Produces(MediaType.APPLICATION_JSON)
    public List<HostStoreInfo> streamsMetadataForStore(
    @PathParam("storeName") String store) {
        return metadataService.streamsMetadataForStore(store);
    }
    @GET
    @Path("{storeName}/value")
    @Produces(MediaType.APPLICATION_JSON)
    public ModelServingInfo servingInfo(
    @PathParam("storeName") final String storeName) {
        // Get the Store
        final ReadableModelStateStore store = streams.store(
          storeName, new ModelStateStore.ModelStateStoreType());
        if (store == null) {
            throw new NotFoundException();
        }
        return store.getCurrentServingInfo();
    }

...
}
```

This implementation provides several REST methods:

- Get the list of application instances (remember, for scalability it is possible to run multiple instances of a Kafka Streams application, where each is responsible for a subset of partitions of the topics). It returns back the list of instances with a list of state stores available in each instance.
- Get a list of application instances containing a store with a given name.
- Get `Model` serving information from a store with a given name.

This code requires implementation of two additional classes: a class used as a data container for returning information about stores on a given host, and a model store type used for locating a store in the Kafka Streams instance. Example 9-10 shows what the data container class looks like (complete code available here).

*Example 9-10. Host store information*

```
public class HostStoreInfo {
   private String host;
```

```
   private int port;
   private Set<String> storeNames;
   ...
   public String getHost() {return host;}
   public void setHost(final String host) {this.host = host;}\
   public int getPort() {return port;}
   public void setPort(final int port) {this.port = port;}
   public Set<String> getStoreNames() {return storeNames;}
   public void setStoreNames(final Set<String> storeNames) {
       this.storeNames = storeNames;
   }
}
```

The model store type looks like Example 9-11 (complete code available here):

*Example 9-11. Host store information*

```
public class ModelStateStoreType implements
QueryableStoreType<ReadableModelStateStore> {
   @Override public boolean accepts(StateStore stateStore) {
       return stateStore instanceof ModelStateStore;
   }
   @Override public ReadableModelStateStore create(
     StateStoreProvider provider,String storeName) {
       return provider.stores(storeName, this).get(0);
   }
}
```

Finally, to bring this all together, you need to update the overall implementation of model serving with Kafka Streams covered in Example 7-6, as shown in Example 9-12 (complete code available here).

*Example 9-12. Updated model serving with Kafka Streams*

```
public class ModelServerWithStore {
   final static int port=8888;

   public static void main(String [ ] args) throws Throwable {
   ...
       // Start the Restful proxy for servicing remote access
       final QueriesRestService restService =
         startRestProxy(streams, port);
       // Add shutdown hook to respond to SIGTERM
       Runtime.getRuntime().addShutdownHook(new Thread(() -> {
           try {
               streams.close();
               restService.stop();
           } catch (Exception e) {
               // ignored
           }
```

```
      }));
  }
  ...
  static QueriesRestService startRestProxy(KafkaStreams streams,
    int port) throws Exception {
      final QueriesRestService restService =
        new QueriesRestService(streams);
      restService.start(port);
      return restService;
  }
}
```

After this is done, you can obtain the state store content by querying the store by name.

## Akka Streams

Akka Streams does not support queryable state (or any state), but by introducing a small change to our custom stage implementation (Example 8-2), it is possible to expose the state from the stage.

To do this, you first must create an interface for querying the state from the stage, as shown in Example 9-13 (compare to Kafka Streams queryable state store interface, Example 9-6).

*Example 9-13. State query interface*

```
trait ReadableModelStateStore {
 def getCurrentServingInfo: ModelToServeStats
}
```

With this in place, you can write stage implementation to support this interface and collect statistics, as demonstrated in Example 9-14 (complete code available here).

*Example 9-14. Updated ModelStage*

```
class ModelStage extends GraphStageWithMaterializedValue
  [ModelStageShape,ReadableModelStateStore] {
    ...
    setHandler(shape.dataRecordIn, new InHandler {
      override def onPush(): Unit = {
        val record = grab(shape.dataRecordIn)
        newModel match {
          ...
        }
```

```
        currentModel match {
          case Some(model) => {
            val start = System.currentTimeMillis()
            val quality = model.score(record.asInstanceOf[AnyVal]).
              asInstanceOf[Double]
            val duration = System.currentTimeMillis() - start
            println(s"Calculated quality - $quality calculated in
              $duration ms")
            currentState.get.incrementUsage(duration)
            push(shape.scoringResultOut, Some(quality))
          }
          case _ => {
            println("No model available - skipping")
            push(shape.scoringResultOut, None)
          }
        }
        pull(shape.dataRecordIn)
      }
    })

...
    // We materialize this value
    val readableModelStateStore = new ReadableModelStateStore() {
      override def getCurrentServingInfo: ModelToServeStats =
        logic.currentState.getOrElse(ModelToServeStats.empty)
    }
    new Tuple2[GraphStageLogic, ReadableModelStateStore]
    (logic, readableModelStateStore)
 }
}
```

In this implementation, the `dataRecordIn` handler is extended to collect execution statistics. Additionally, implementation of the State query interface (Example 9-13) is provided so that the stage can be queried for the current model state.

For REST interface implementation, I used Akka HTTP. The resource used for statistics access can be implemented as shown in Example 9-15 (complete code available here).

*Example 9-15. Implementing REST resource*

```
object QueriesAkkaHttpResource extends JacksonSupport {

 def storeRoutes(predictions: ReadableModelStateStore) :
   Route = pathPrefix("stats"){
   pathEnd {
     get {
```

```
      val info: ModelToServeStats =
        predictions.getCurrentServingInfo
      complete(info)
    }
  }
 }
}
```

To bring it all together, you must modify the Akka model server (Example 8-3), as demonstrated in Example 9-16 (complete code available here).

*Example 9-16. Updated Akka model server implementation*

```
object AkkaModelServer {
...

 def main(args: Array[String]): Unit = {

...

   val modelStream: Source[ModelToServe, Consumer.Control] =

...

   val dataStream: Source[WineRecord, Consumer.Control] =

...

   val model = new ModelStage()

   def keepModelMaterializedValue[M1, M2, M3](
     m1: M1, m2: M2, m3: M3): M3 = m3

   val modelPredictions :
     Source[Option[Double], ReadableModelStateStore] =
    Source.fromGraph(
     GraphDSL.create(dataStream, modelStream, model)(
       keepModelMaterializedValue) {
       implicit builder => (d, m, w) =>
         import GraphDSL.Implicits._

         d ~> w.dataRecordIn
         m ~> w.modelRecordIn
         SourceShape(w.scoringResultOut)
     }
   )

 val materializedReadableModelStateStore: ReadableModelStateStore =
     modelPredictions
       .map(println(_))
```

```
      .to(Sink.ignore)
      .run() // run the stream, materializing the StateStore

  startRest(materializedReadableModelStateStore)
 }

 def startRest(service : ReadableModelStateStore) : Unit = {

  implicit val timeout = Timeout(10 seconds)
  val host = "localhost"
  val port = 5000
  val routes: Route = QueriesAkkaHttpResource.storeRoutes(service)

  Http().bindAndHandle(routes, host, port) map
    { binding => println(s"REST interface bound to
       ${binding.localAddress}") }
    recover { case ex => println(
       s"REST interface could not bind to $host:$port", ex.getMessage)
  }
 }
}
```

There are several changes here:

- Instead of using `dropMaterializedValue`, we are going to use `keepModelMaterializedValue`.
- A new method `startRest` is implemented, which starts an internal REST service based on the resource (Example 9-16) and the implementation of the interface (Example 9-14).
- Materialized state is used for accessing statistics data.

Although this solution provides access to local (instance-based) model serving statistics, it does not provide any support for getting application-based information (compare to the queryable Kafka Streams store). Fortunately Kafka itself keeps track of instances with the same group ID and provides (not very well documented) AdminClient APIs (usage example) with which you can get the list of hosts for a consumer group. Assuming that all instances execute on different hosts with the same port, you can use this information to discover all applications instances and connect to all of them to get the required information. This is not a completely reliable method, but you can use it in the majority of cases to get complete application statistics.

## Spark and Beam

Neither Spark nor Beam currently support queryable state, and I have not seen any definitive plans and proposals to add this support in the future. So, if you use either of these tools, you can implement monitoring by using either logging or an external database, for example, Cassandra.

In the case of Spark, there is an additional option: use the Spark Job Server, which provides a REST API for Spark jobs and contexts. The Spark Job Server supports using Spark as a query engine (similar to queryable state).

Architecturally Spark Job Server consists of a REST job server providing APIs to consumers and managing application jars, execution context, and job execution on the Spark runtime. Sharing contexts allows multiple jobs to access the same object (the Resilient Distributed Dataset [RDD] state in our case). So, in this case, our Spark implementation (Example 6-1) can be extended to add a model execution state to the RDD state. This will enable creation of a simple application that queries this state data using Spark Job Server.

## Conclusion

You should now have a thorough understanding of the complexities of serving models produced by machine learning in streaming applications. You learned how to export trained models in both TensorFlow and PMML formats and serve these models using several popular streaming engines and frameworks. You also have several solutions at your fingertips to consider. When deciding on the specific technology for your implementation, you should take into account the number of models you're serving, the amount of data to be scored by each model and the complexity of the calculations, your scalability requirements, and your organization's existing expertise. I encourage you to check out the materials referenced throughout the text for additional information to help you implement your solution.

## About the Authors

**Boris Lublinsky** is a Principal Architect at Lightbend. He has over 25 years of experience in enterprise, technical architecture, and software engineering. He is coauthor of *Applied SOA: Service-Oriented Architecture and Design Strategie*s (Wiley) and *Professional Hadoop Solutions* (Wiley). He is also an author of numerous articles on architecture, programming, big data, SOA, and BPM.