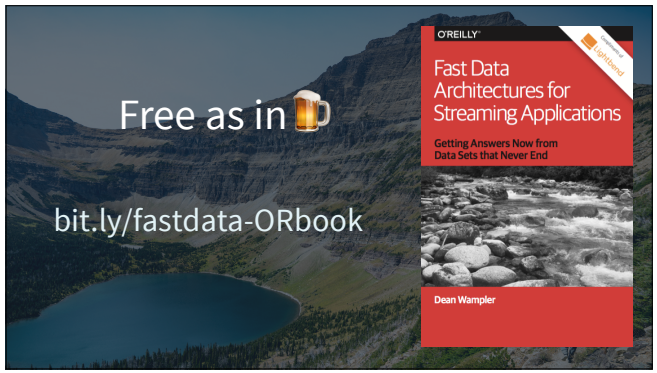
You can download this and my other talks from the polyglotprogramming.com/talks link.

Photo: Oldman Lake at Sunrise. All photos are from Glacier National Park, Montana, USA.

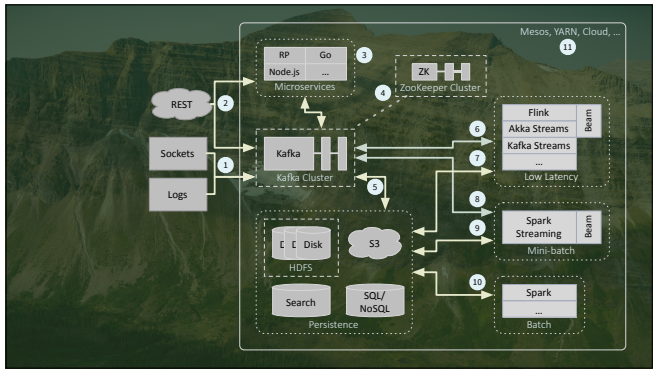
Photo: Descending to Oldman Lake from Pitamakan Pass, the day before, Glacier National Park, Montana, USA.



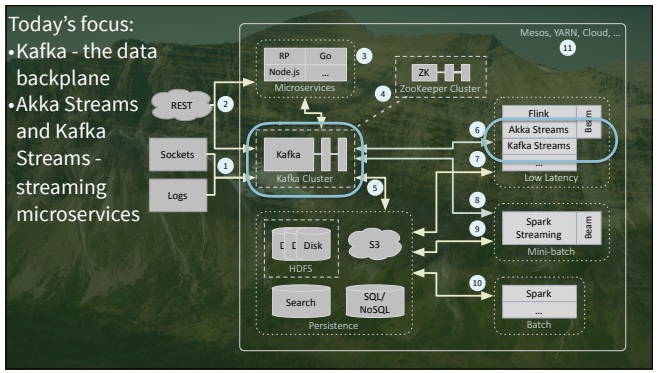
My book, published last 2016, that describes my view about streaming architectures. We'll drill into a piece of it in this talk.

Photo: Full view of Mt Phillips with Lupfer Glacier, from just North of Dawson Pass

---



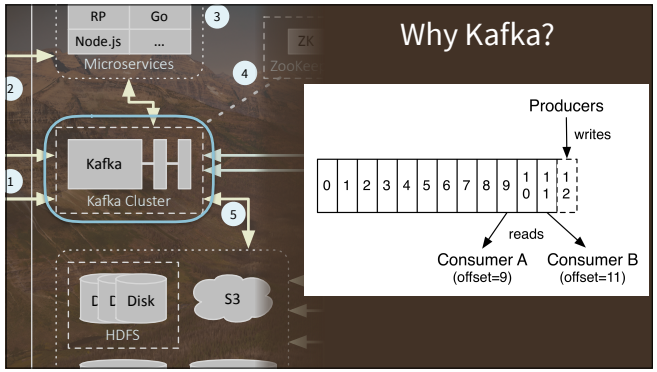
This architecture diagram is taken from the report. The numbers correspond to sections in the report. I'll quickly go through this diagram (not following the order of the numbers ;^), for context, then focus on the streaming engines on the right, the focus of today's talk.



Kafka is the data backplane for high-volume data streams, which are organized by topics. Kafka has high scalability and resiliency, so it's an excellent integration tool between data producers and consumers.

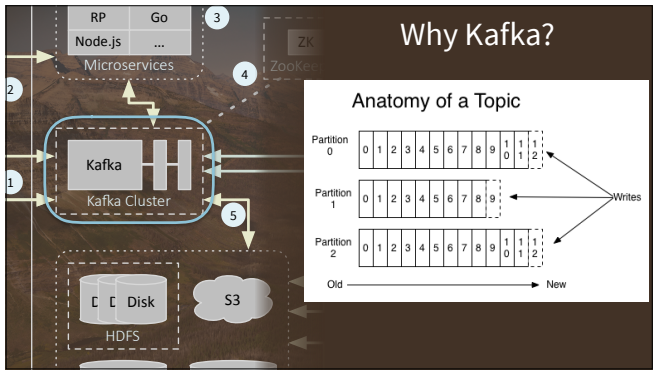
Photo: L to R, Mt Pinchot, Mt Stimson, Blackfoot Mtn with Pumpelly Glacier, and Tickham Mtn, with burns around Nyak Creek

---



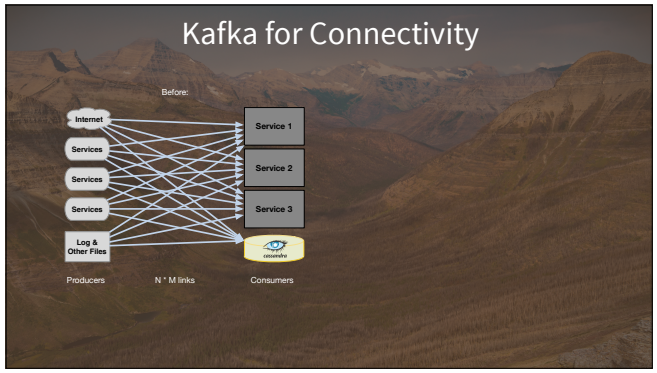
Kafka is a distributed log, storing messages sequentially. Producers always write to the end of the log, consumers can choose the log offset where they want to read from (earliest, latest, etc.)
Kafka is not a traditional queue, where entries are popped when read.

Alternatives to Kafka include Pravega (EMC) and DistributedLog/Pulsar (Apache)
Image: Apache Kafka website

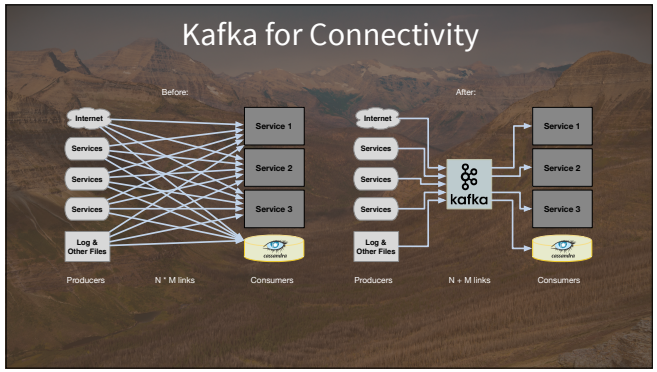


Data is organized by topic, which can have 1 or more partitions for scalability through parallelism. Partitions can be replicated for resiliency. They are the physical data storage artifact. Data in a partition is guaranteed to be sequential; not true for topics! (Unless they have only 1 partition…)

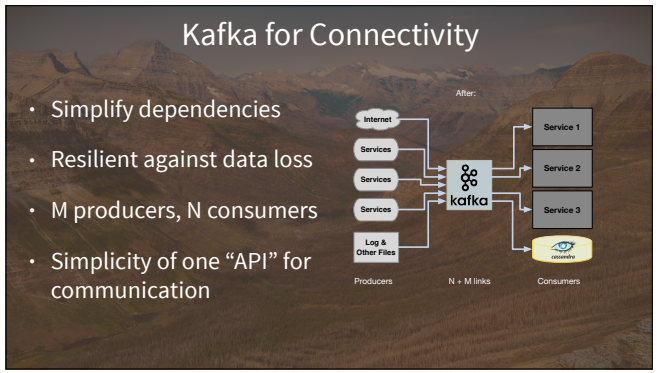
Image: Apache Kafka website



Several problems here: 1) Services are coupled! 2) What if Service 1 crashes; we might lose data from all the upstream producers connected to it. 3) Every producer-consumer pair has to understand the API and behavior of its "peer". 4) It's hard to understand what's going on. But we know we can solve any problem in computer science with another level of indirection…

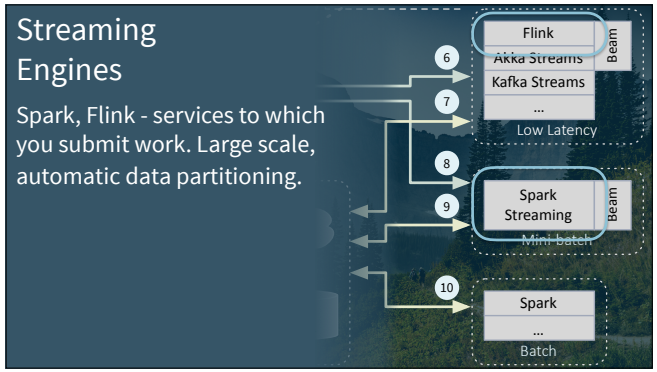


Kafka simplifies the dependencies between services, provides robustness when a service crashes (data is captured safely, waiting for the service to be restarted), minimizes writes, and provides the simplicity of one "API" for communication between services.

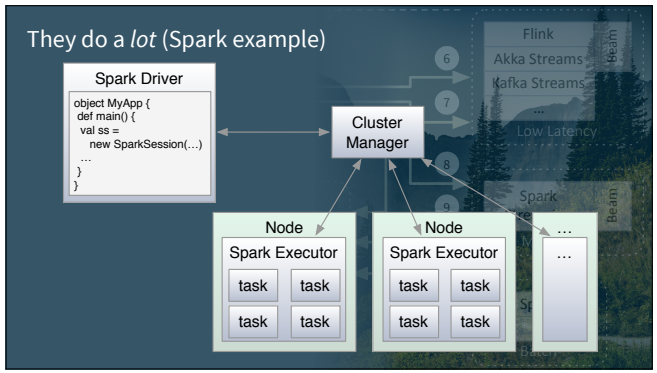


Kafka simplifies the dependencies between services, provides robustness when a service crashes (data is captured safely, waiting for the service to be restarted), minimizes writes (i.e., write once to a topic, not N times to N consumers), and provides the simplicity of one "API" for communication between services.

Photo: Rounding a Corner on the Iceberg Lake Trail

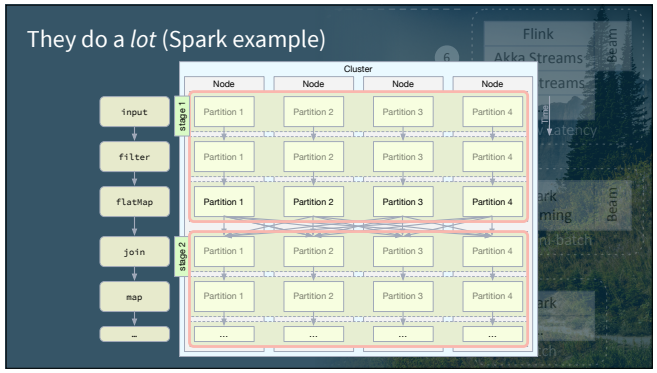


They support highly scalable jobs, where they manage all the issues of scheduling processes, etc. You submit jobs to run to these running daemons. They handle scalability, failover, load balancing, etc. for you.



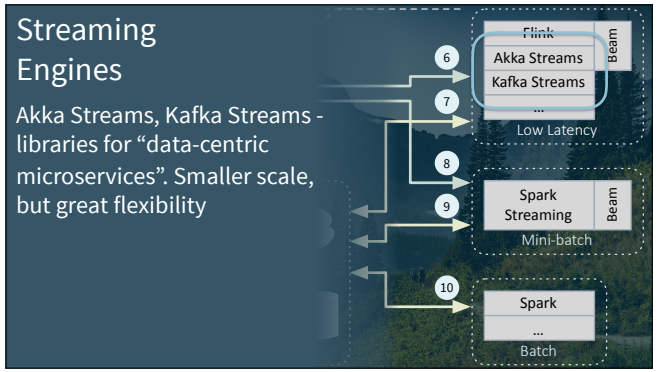
You have to write jobs, using their APIs, that conform to their programming model. But if you do, Spark and Flink do a great deal of work under the hood for you!

You submit your job to a cluster manager, which breaks it down into tasks to run on nodes around the cluster.



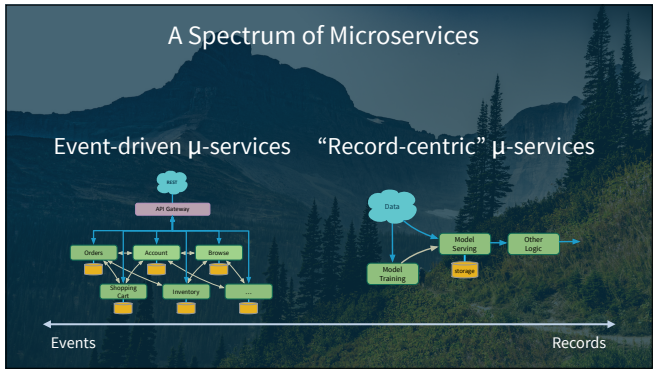
You have to write jobs, using their APIs, that conform to their programming model. But if you do, Spark and Flink do a great deal of work under the hood for you!

An example of how Spark decomposes your logical data flow or query into “stages” each of which has one JVM per data “partition”. Spark also handles partitioning for you. Flink works in a similar way.

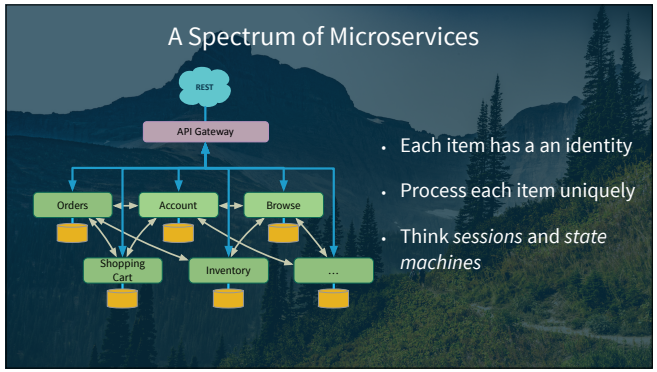


Much more flexible deployment and configuration options, compared to Spark and Flink, but more effort is required by you to run them. They are “just libraries”, so there is a lot of flexibility and interoperation capabilities.

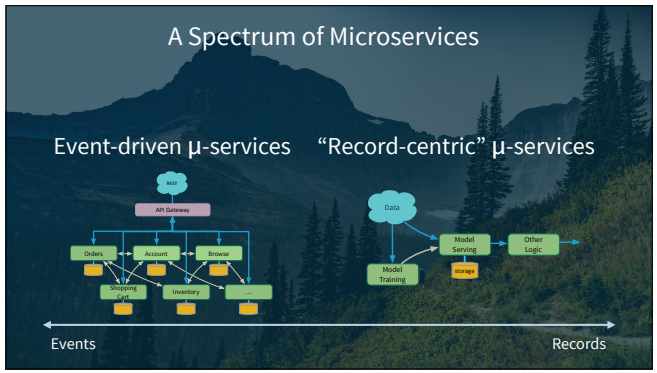
https://twitter.com/shanselman/status/967703711492423682



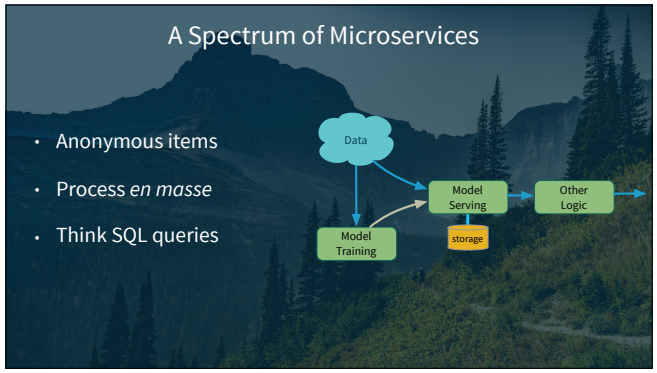
When comparing Akka Streams vs. Kafka Streams, I think it helps to consider their legacies. To do this, it’s useful to consider a spectrum of microservices, *event-driven* vs. *record-centric.*



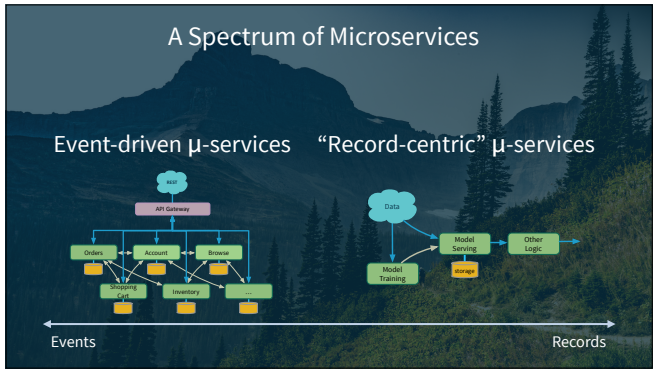
By event-driven microservices, I mean that each individual datum is treated as an event that triggers some activity, like steps in a shopping session. Each event requires individual handling, routing, responses, etc. REST, CQRS, and Event Sourcing are ideal for this. Think how a browser session works, such as shopping on an ecommerce site. Or, think about state machines.



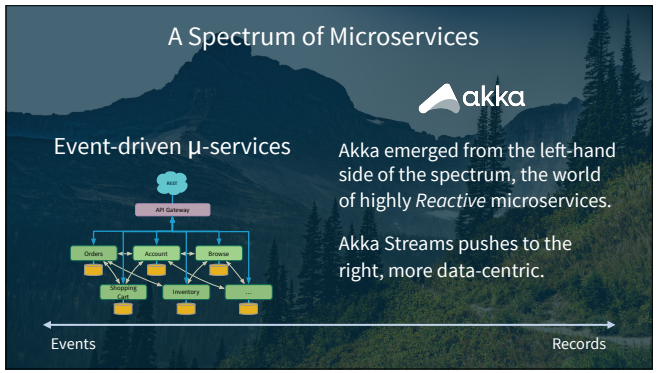
(transition slide)



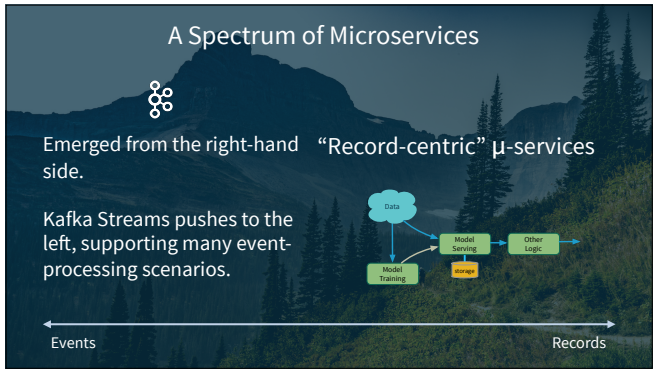
On the other end, records are *anonymous* in this picture; we process them as a group, for efficiency, and don't typically examine them individually.
I know that records in SQL databases often have unique ids that you can query individually, but we're talking about stream processing. Also, it's a spectrum, because we might take events and also route them through a data pipeline, like computing statistics or scoring against a machine learning model (as suggested here), perhaps for fraud detection, recommendations, etc.



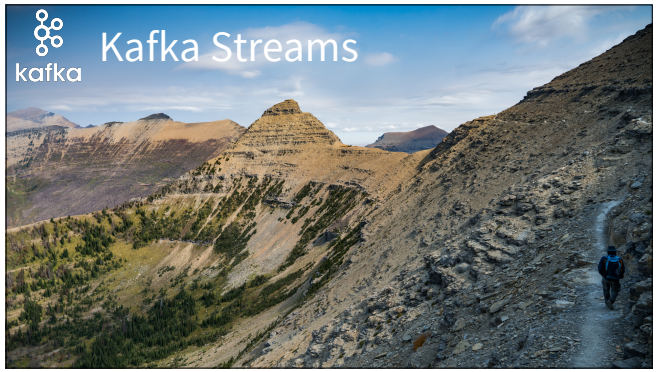
(transition slide)



Akka *Actors* emerged in the world of building *Reactive* microservices, those requiring high resiliency, scalability, responsiveness, and CEP, that must be event driven. Akka is extremely lightweight and supports extreme parallelism, including across a cluster, and low latency, all without having to do threadsafe programming yourself. However, the Akka Streams API is effectively a dataflow API, so it nicely supports many streaming data scenarios, when the Actor model is too low level.

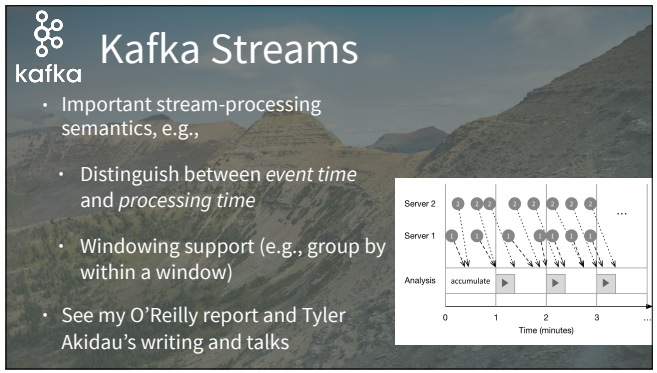


Kafka reflects the heritage of moving and managing streams of data, first at LinkedIn. Certainly the features of Kafka Streams are focused on record processing, as we'll discuss. But again, from the beginning, Kafka has been used for event-driven microservices, too, where the "stream" contained events, rather than records. So, while Kafka Streams fits squarely in the record-processing world, where you define data flows for processing and even SQL, it can also be used for event processing scenarios. I.e., it moves to the left on the spectrum…

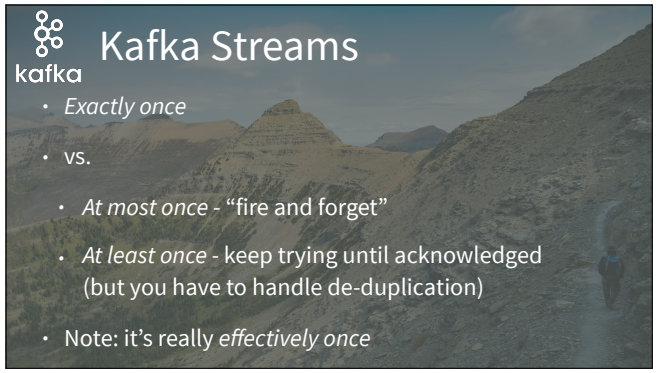


Let's start our "journey" to explore Kafka Streams

Photo: Starting towards Cut Bank Pass

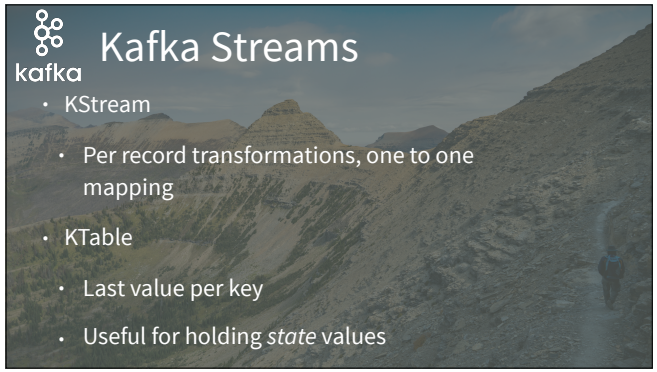


There's a maturing body of thought about what streaming semantics should be, too much to discuss here. Dean's book provides the next level of details. See Tyler's work (from the Google Apache Beam team) for deep dives. He's also speaking at this conference!

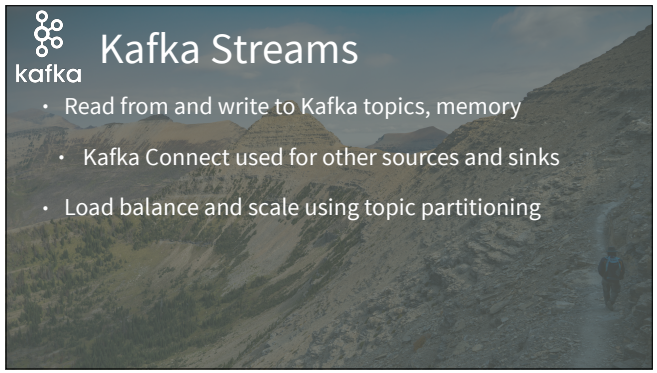


KS offers "exactly once" processing, but theoretically, it's impossible to eliminate all failure scenarios, so "effectively once" is used to describe systems that are very close to "exactly once".

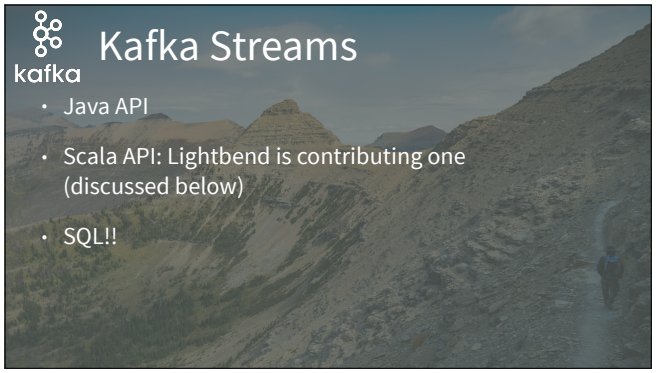
See Karthik Ramasamy's talk right after mine on this concept: https://conferences.oreilly.com/strata/strata-ca/public/schedule/detail/64107



There is a duality between streams and tables. Tables are the latest state snapshot, while streams record the history of state evolution. A common way to implement databases is to use an event (or change) log, then update the state from the log.

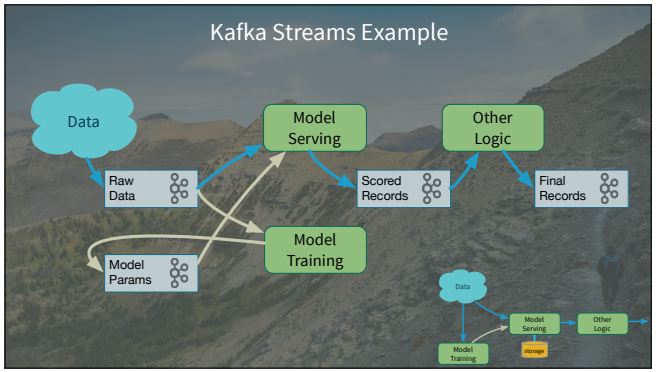


KS is very Kafka-centric; it doesn't try to support connecting to other sources and sinks, but you can use Kafka Connect for that purpose. Scalability and load balancing is implemented by you; how you decide to partition your topics.

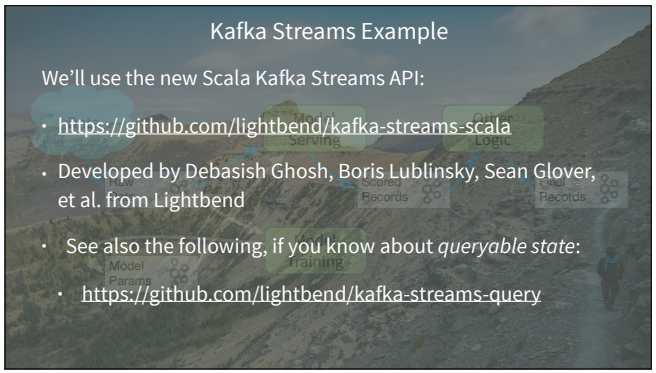


The kafka-streams-query uses a KS API to find all the partitions across a cluster for a given topic, query their state, and aggregate the results, behind a web service. Otherwise, you have to query the partitions individually yourself.
The SQL interface actually uses a service, rather than an API (like Spark or Flink). Adding SQL so soon to a young product is another indication that KS comes from the data side of the service spectrum

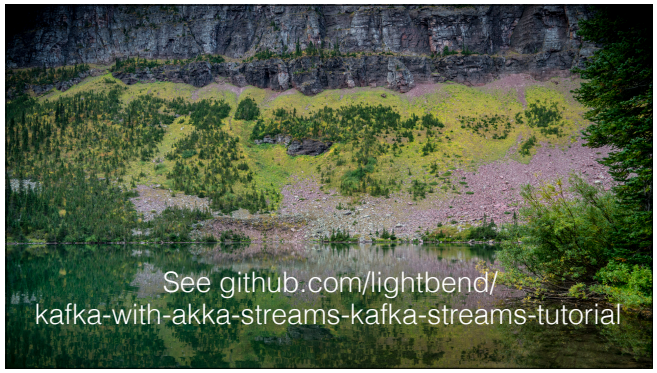
---



Here's a streaming microservice example for Kafka Streams, where we will use Kafka topics as intermediate "queues". We'll show code for the "Model Serving" microservice. The "Model Training" could be implemented with another system, like SparkML, TensorFlow, etc. The "Other Logic" service would do further processing, but we don't need to look at it here to get the idea…

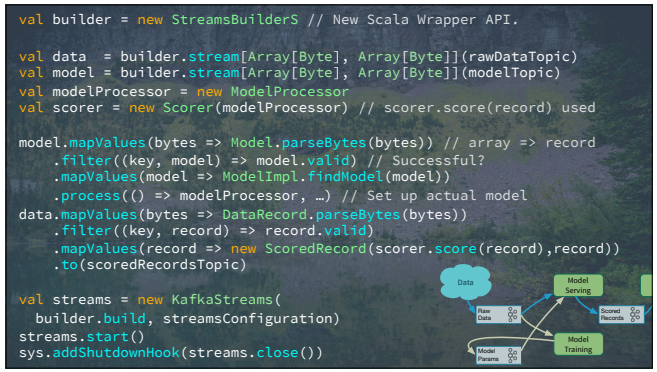


Our example will use the new Scala-based Kafka Streams API (https://github.com/lightbend/kafka-streams-scala), developed by my colleagues on the Fast Data Platform project. It adheres very closely to the semantics of the Java API, i.e., it tries to be a thin wrapper, rather than idiomatic Scala (no Monads…). Lightbend is contributing this API to Apache Kafka.

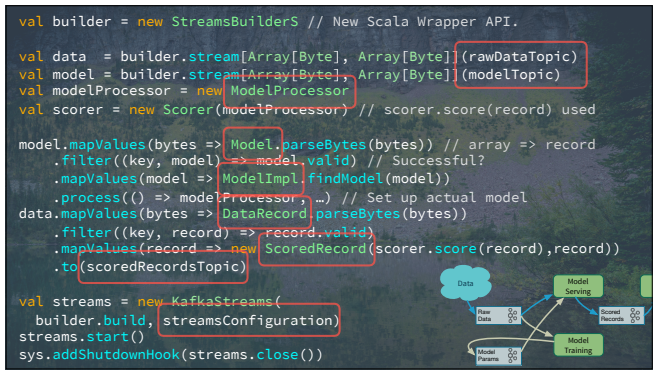


The code examples today are loosely based on the tutorial code that Boris Lublinsky and I presented earlier today. I won't have time to explain all the details. The goal is to give you the "gist" of these APIs.

Photo: Morning at No Name Lake

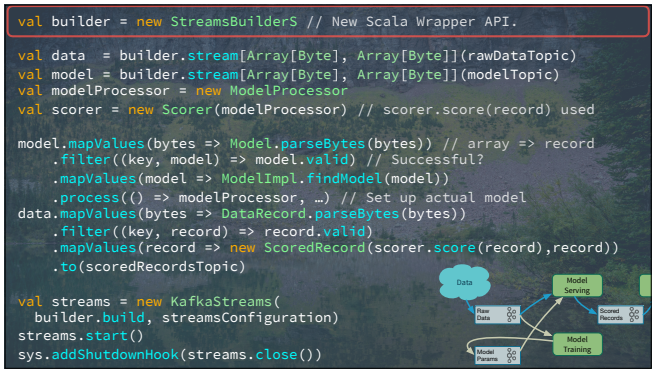


Example using the new Scala-based Kafka Streams API (https://github.com/lightbend/kafka-streams-scala). It adheres very closely to the semantics of the Java API. Lightbend is contributing this API to Apache Kafka. We'll sketch the implementation of the diagram parts shown, but not fill in all the details. This example is adapted from the tutorial at forthcoming conferences I'll mention at the end.

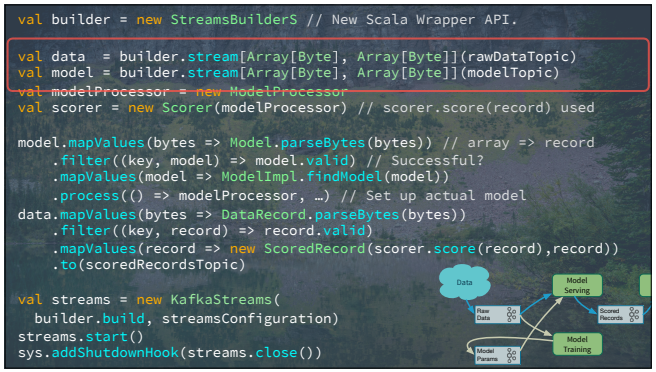


For time, I won't show these definitions, but just describe them briefly.

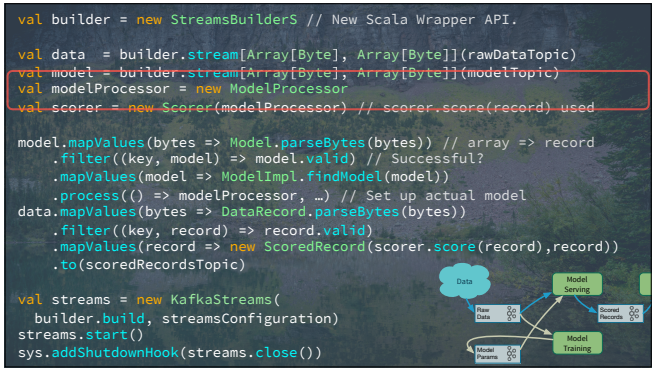
- Not shown are import statements and configuration settings (streamsConfiguration) for Kafka broker locations, etc.
- rawDataTopic, modelTopic, and scoredRecordsTopic are names of Kafka topics.
- ModelProcessor knows how to update the in-memory model when new parameters arrive.
- Model is an abstraction for models and knows how to deserialize bytes into model parameters.



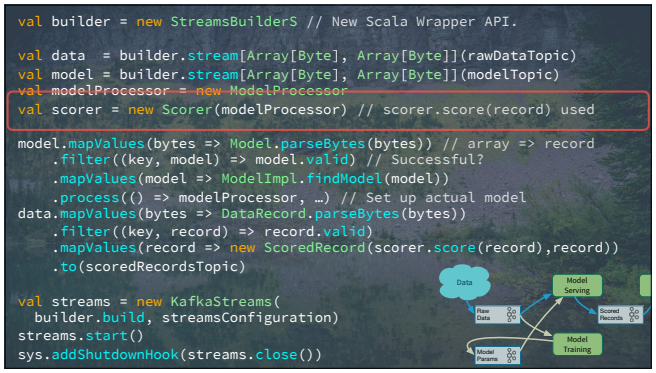
The Scala API follows the Builder Pattern used by the Java API, the entry point for constructing KS apps. Note the “S” suffix; this is the convention used in the Scala API for the Java API types it wraps.



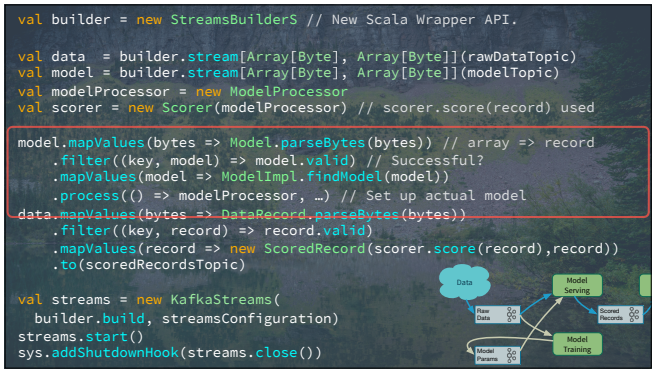
Create two input streams, where the keys and values are “raw” byte arrays. One is for the data feed and the other is for the parameters for the model to use for scoring the data.



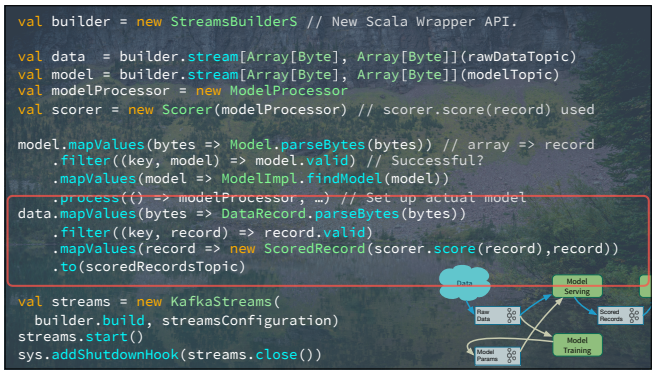
Handle instantiating the model (such as TensorFlow vs. …) and updating when new parameters (model updates) are available.



Will take a record, score it with the current model (call "scorer.score(record)"), then return a new record that includes the score (as shown below).



For the stream of model parameters, a byte array, first parse it into a Model object, then filter for only those results that were successfully parsed ("valid"). (Ignoring bad results, but you could send those down a different path…) Then use those Model objects to find an implementation supported by the system, then use a "side-effecting" Processor to handle the effort of instantiating the implementation, loading the parameters to it, etc. That the "modelProcessor" was given to the Scorer instance, which will use the processor for scoring.



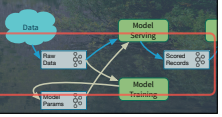
For the data stream, also a byte array, parse it into a DataRecord object, then filter for only those that were successfully parsed ("valid"), ignoring errors. Now we use or scorer to score the record, passing the result as the first argument to a new record type, ScoredRecord, along with the old record. Finally, we write these new records to a new Kafka topic.

```
val builder = new StreamsBuilderS // New Scala Wrapper API.

val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
    .filter((key, model) => model.valid) // Successful?
    .mapValues(model => ModelImpl.findModel(model))
    .process(() => modelProcessor, …) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
    .filter((key, record) => record.valid)
    .mapValues(record => new ScoredRecord(scorer.score(record),record))
    .to(scoredRecordsTopic)

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())
```



Next to last steps, take the builder and the streams we've defined, and constructs our KS system, also passing a configuration object (not shown). Finally we start processing and run forever!

```
val builder = new StreamsBuilderS // New Scala Wrapper API.

val data  = builder.stream[Array[Byte], Array[Byte]](rawDataTopic)
val model = builder.stream[Array[Byte], Array[Byte]](modelTopic)
val modelProcessor = new ModelProcessor
val scorer = new Scorer(modelProcessor) // scorer.score(record) used

model.mapValues(bytes => Model.parseBytes(bytes)) // array => record
    .filter((key, model) => model.valid) // Successful?
    .mapValues(model => ModelImpl.findModel(model))
    .process(() => modelProcessor, …) // Set up actual model
data.mapValues(bytes => DataRecord.parseBytes(bytes))
    .filter((key, record) => record.valid)
    .mapValues(record => new ScoredRecord(scorer.score(record),record))
    .to(scoredRecordsTopic)

val streams = new KafkaStreams(
  builder.build, streamsConfiguration)
streams.start()
sys.addShutdownHook(streams.close())
```

Clean up your mess! When the system shutdowns, properly close the streams.

What's Missing?

Kafka Streams is a powerful library, but you'll need to provide the rest of the microservice support telling through other means. (Some are provided if you run the support services for the SQL interface.)

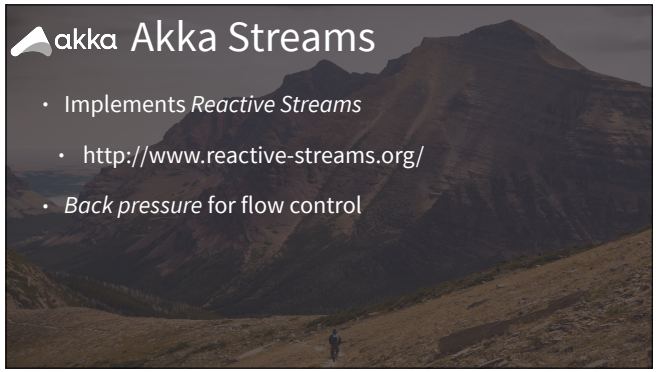
You would embed your KS code in microservices written with more comprehensive toolkits, such as the Lightbend Reactive Platform!

We'll return to this point…

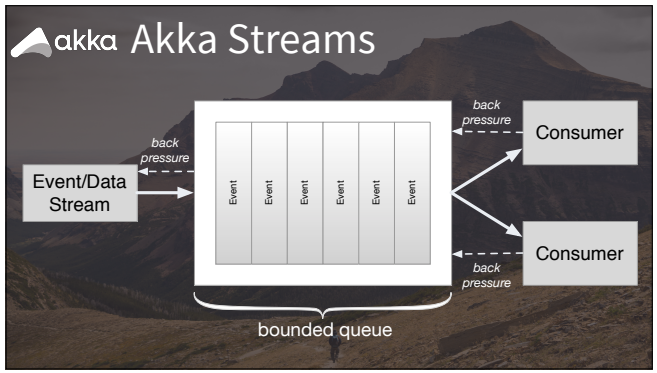
Let's continue our "journey" to explore Akka Streams

Photo: Descending to Pitamaken Pass, with Rising Wolf Mtn and Lower Two Medicine Lake and Montana prairie beyond

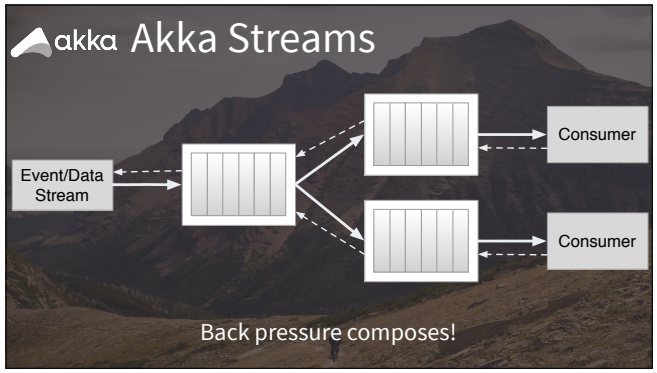
---



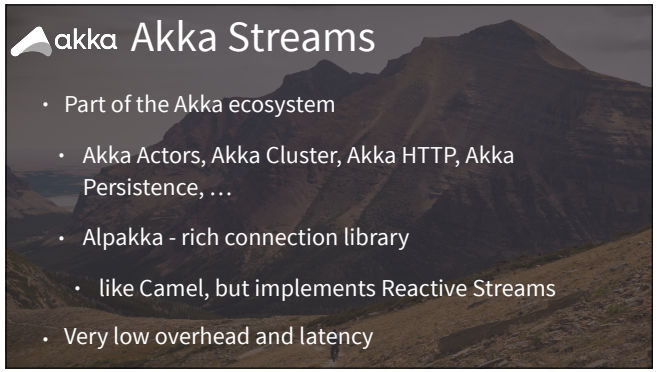
See this website for details on why *back pressure* is an important concept for reliable flow control, especially if you don't use something like Kafka as your "near-infinite" buffer between services.



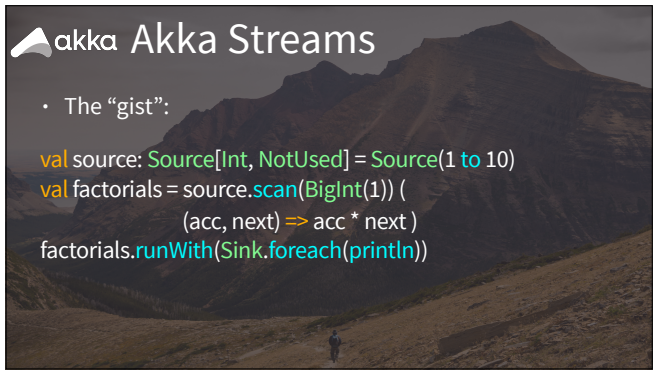
Bounded queues are the only sensible option (even Kafka topic partitions are bounded by disk sizes), but to prevent having to drop input when it's full, consumers signal to producers to limit flow. Most implementations use a push model when flow is fine and switch to a pull model when flow control is needed.



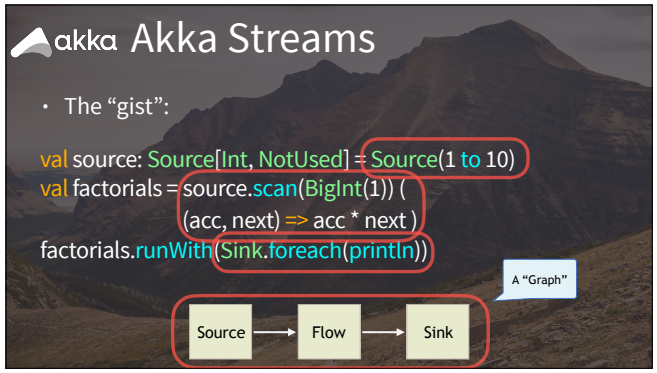
And they compose so you get end-to-end back pressure.



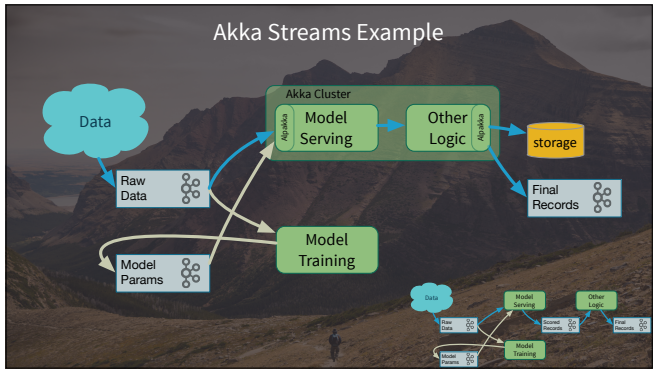
Rich, mature tools for the full spectrum of microservice development. Akka Streams adds a streaming abstraction on top of Akka actors. It's ideal when complex event processing (CEP) is the preferred model, as opposed to in bulk processing of data. Akka's powerful Actor model abstracts over the details of thread programming for highly concurrent apps, with libraries for clustering, persisting state, and data interchange with many sources and sinks (the "Alpakka" project).



A very simple example to illustrate the Akka Stream model.



Rich, mature tools for the full spectrum of microservice development.



Here’s our streaming microservice example adapted for Akka Streams. We’ll still use Kafka topics in some places and assume we’re using the same implementation for the “Model Training” microservice. Alpakka provides the interface to Kafka, DBs, file systems, etc. We’re showing two microservices as before, but this time running in Akka Cluster, with direct messaging between them. We’ll explore this a bit more after looking at the example code.

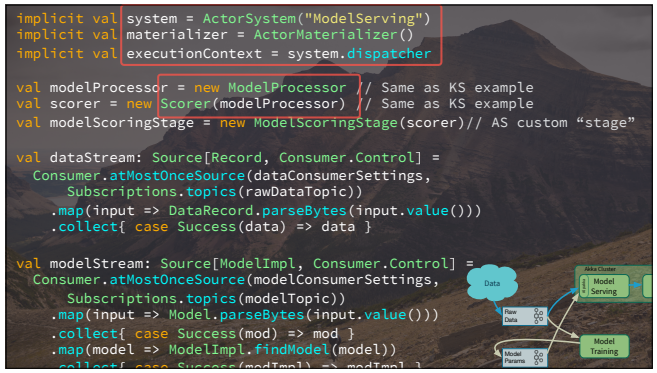
```
implicit val system = ActorSystem("ModelServing")
implicit val materializer = ActorMaterializer()
implicit val executionContext = system.dispatcher

val modelProcessor = new ModelProcessor // Same as KS example
val scorer = new Scorer(modelProcessor) // Same as KS example
val modelScoringStage = new ModelScoringStage(scorer)// AS custom “stage”

val dataStream: Source[Record, Consumer.Control] =
  Consumer.atMostOnceSource(dataConsumerSettings,
      Subscriptions.topics(rawDataTopic))
    .map(input => DataRecord.parseBytes(input.value()))
    .collect{ case Success(data) => data }

val modelStream: Source[ModelImpl, Consumer.Control] =
  Consumer.atMostOnceSource(modelConsumerSettings,
      Subscriptions.topics(modelTopic))
    .map(input => Model.parseBytes(input.value()))
    .collect{ case Success(mod) => mod }
    .map(model => ModelImpl.findModel(model))
```

This example is longer, not quite fitting on a page. It has a little more detail than the previous one, and the Akka Streams API offers more options to chose from for processing. As before, this code is loosely based on the tutorial mentioned previously.

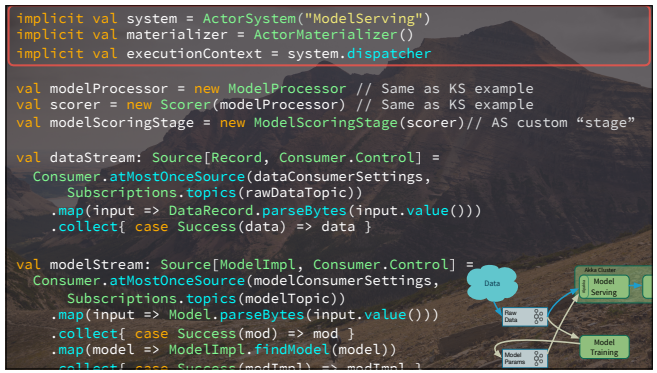


As before, we won't show all the code:

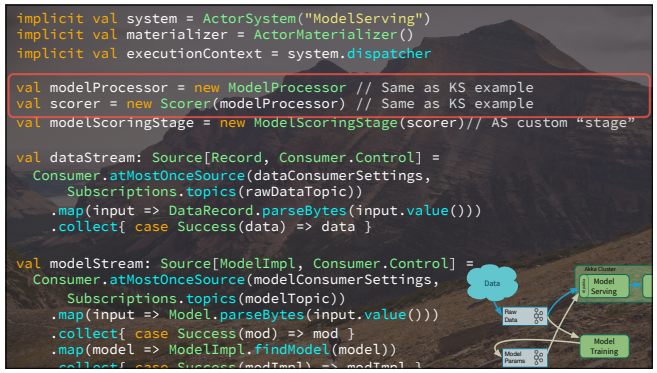
* ActorSystem is the entry point for all Akka libraries.
* ActorMaterializer instantiates our graphs of "Flows" using Actors.
* An ExecutionContext encapsulates how threading in managed internally.
* We'll reuse ModelProcessor and Scorer from before.

As before, we won't show all the code (continued):

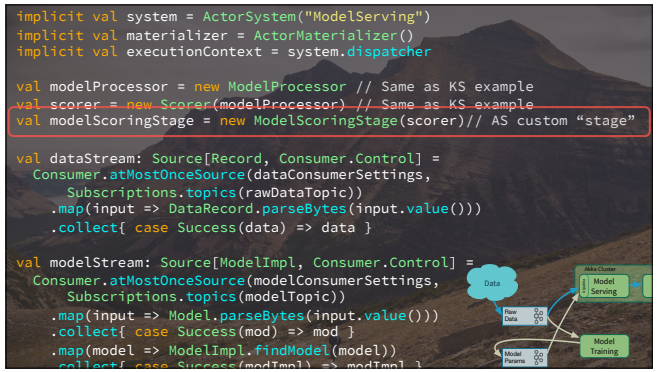
* Source and Sink we described before; general AS concepts.
* Consumer*, Producer*, and Subscriptions are AS APIs for Kafka Consumer and Producer APIs, etc.
* dataConsumerSettings (and similar below) encapsulate configuration, like broker locations, etc.
* Success is used to hold a result where an exception wasn't thrown. A Failure would wrap the exception.



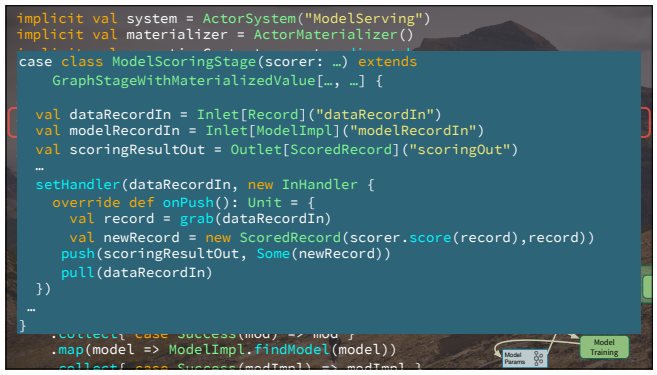
This is how Akka Systems are initialized and the default way to materialized Akka Streams graphs. Akka Streams separates the logical definitions and "physical" materializations of streams.



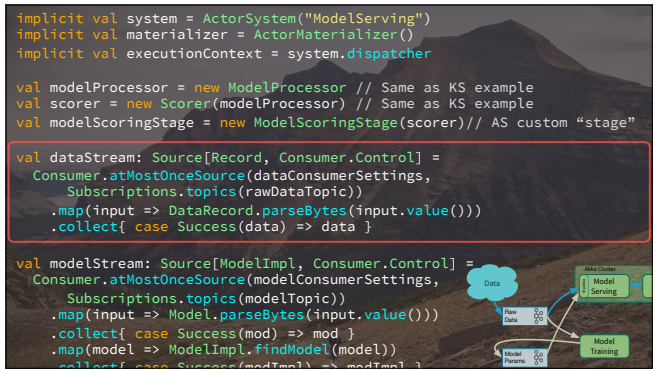
Same as in the KS example.



You can define custom "stages" in AS for special processing. It's not strictly necessary here; we could use a map over the data stream, but it shows an option for encapsulating complex logic, when necessary.



A sketch of the custom stage. It's a graph node, with zero or more "Inlets" (2) and zero or more "Outlets" (1). We set up a callback handler that's called for each record available (model updates not shown). As for the KS example, when processing data records, it scores the record with the current model, then constructs a new record to push out, then signals through "pull" that it's ready for the next record.



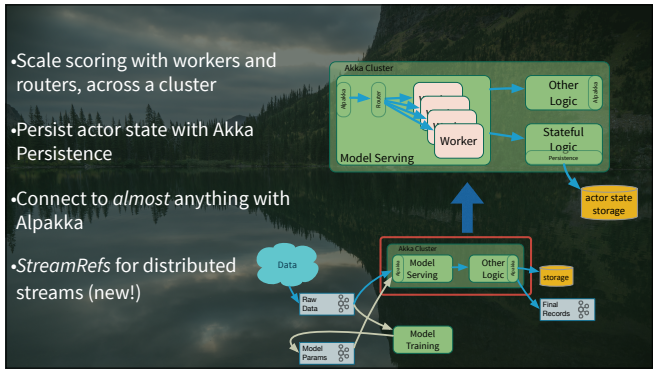
Recall the KS example; here is the AS equivalent for handling the incoming data records, as byte arrays that need to be parsed. Note the "at least once" semantics. The "collect { pattern match }" idiom is like a "filter", except we want to change what's returned ("filter" returns elements of the same input type.) So, if the returned element is a "Success(data)", the parsing succeeded (otherwise a "Failure(exception)" is returned). We return the "data". "dataConsumerSettings" is an Akka API ConsumerSettings object that defines the connection to the Kafka topic (definition not shown). In AS, Sources and Sinks are abstract; we can glue these "streamlets" together to

---

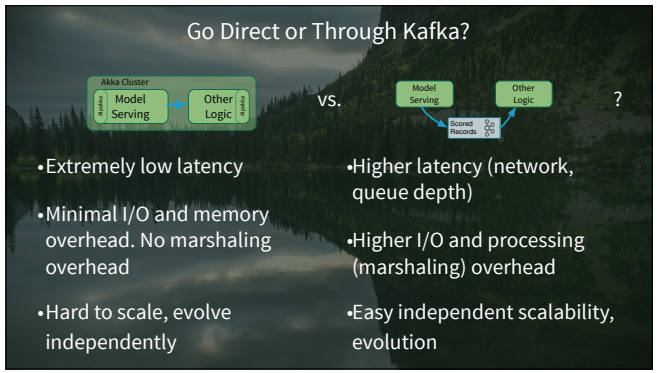
Similarly, process the model parameters, then instantiate the correct model, and set the new model in the modelProcessor object, checking for success as we go.
The last line adds a "no-op" Sink (there's nothing to output), and then runs this stream.

Take the data stream, materialize the custom scoring stage (this is done implicitly). The "left" and "right" correspond to "dataStream" and "modelScoringStage" in the general case. Here, we need to keep the right stream output by the custom stage, which as implemented (not shown) will return our ScoredRecord objects. Finally, we convert to an AS wrapper for records published to Kafka and run with a producer that will publish the records to Kafka (configured with some publisherSettings - not shown)

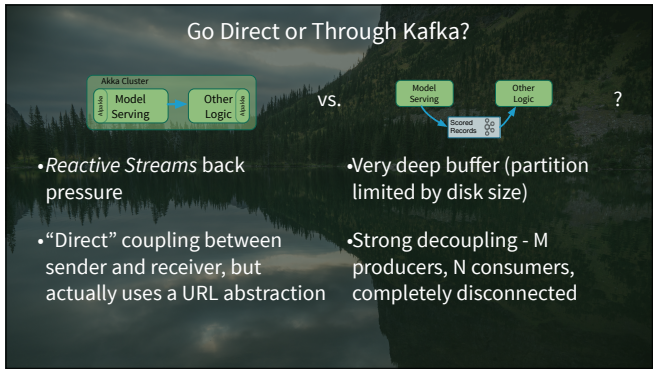
Photo: Morning at No Name Lake



Here's our streaming microservice example adapted for Akka Streams. We'll still use Kafka topics in some places and assume we're using the same implementation for the "Model Training" microservice. Alpakka provides the interface to Kafka, DBs, file systems, etc. We're showing two microservices as before, but this time running in Akka Cluster, with direct messaging between them. We'll explore this a bit more after looking at the example code. StreamRefs are very new; they make it straightforward to connect streams across JVM boundaries, even machines.



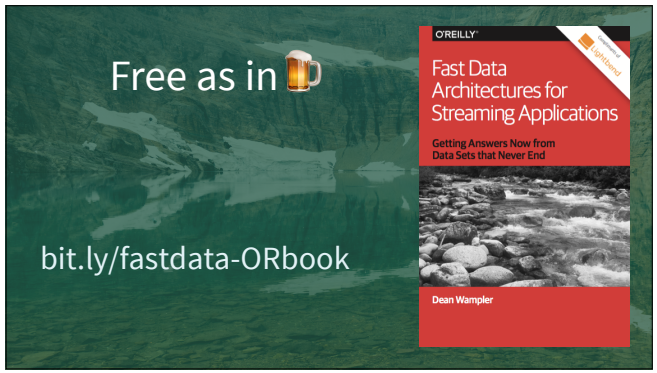
Design choice: When is it better to use direct actor-to-actor (or service-to-service) messaging vs. going through a Kafka topic?



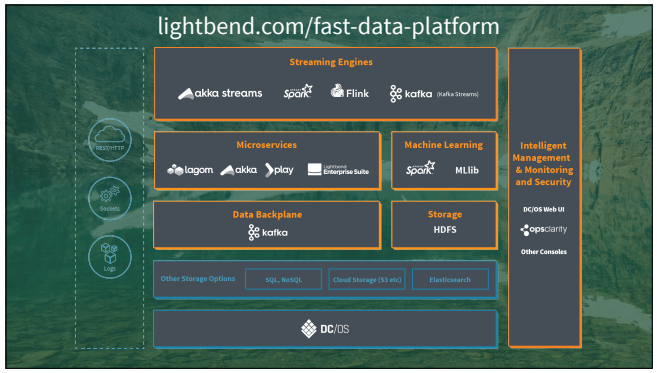
Design choice: When is it better to use direct actor-to-actor (or service-to-service) messaging vs. going through a Kafka topic?

Wrapping up…

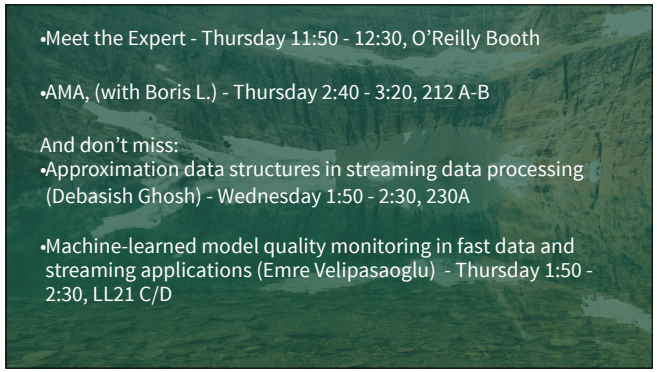
Photo: Iceberg Lake



See my report for more details.



Lightbend created the Fast Data Platform to provide an integrated, tested suite of tools for building, deploying, and running a spectrum of streaming applications using Kafka, Akka Streams, Kafka Streams, as well as Spark and Flink. HDFS is included, too, along with Lightbend's *Reactive Platform* for full-spectrum microservices.



Check out the other Lightbend sessions this week. Also, Boris Lublinsky and I did a tutorial on this material earlier this week. Find it on the conference Safari site after the conference.

Go to lightbend.com/fast-data-platform to learn more about the project I lead to build and run streaming data + microservice systems. You can also download my O'Reilly report from that page.
You can download this and all my other talks from the polyglotprogramming.com/talks link.

Photo: Iceberg Lake, Glacier National Park, Montana, USA.