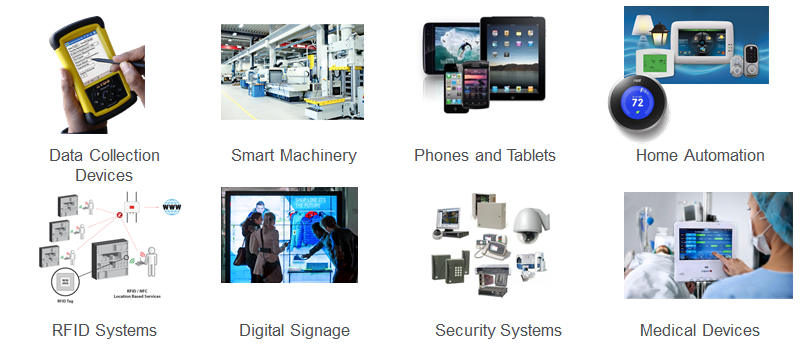
## Fast, Scalable, Streaming Applications with MapR Streams, Spark Streaming, and MapR-DB



April 22, 2016

[Carol McDonald](https://www.mapr.com/blog/author/carol-mcdonald)

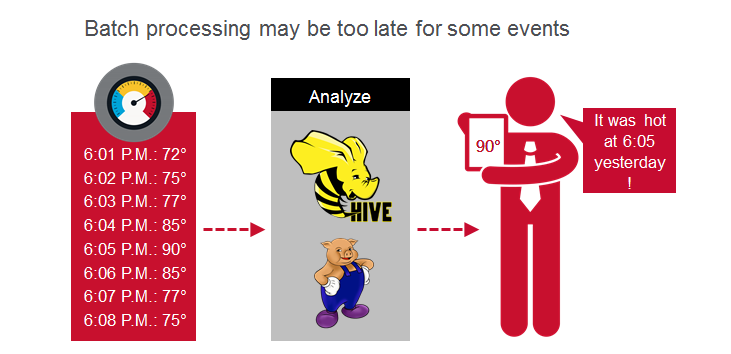
Many of the systems we want to monitor happen as a stream of events. Examples include event data from web or mobile applications, sensors, or medical devices.



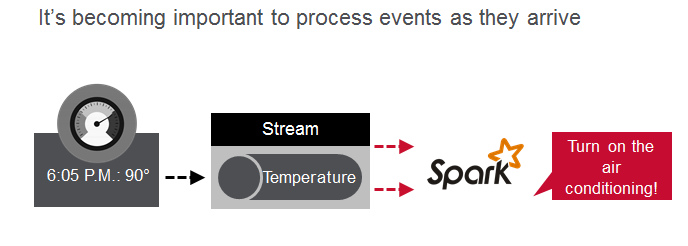
Real-time analysis examples include:

* Website monitoring , Network monitoring
* Fraud detection
* Web clicks
* Advertising
* Internet of Things: sensors

Batch processing can give great insights into things that happened in the past, but it lacks the ability to answer the question of "what is happening right now?”



It is becoming important to process events as they arrive for real-time insights, but high performance at scale is necessary to do this. In this blog post, I'll show you how to integrate Apache Spark Streaming, MapR-DB, and MapR Streams for fast, event-driven applications.

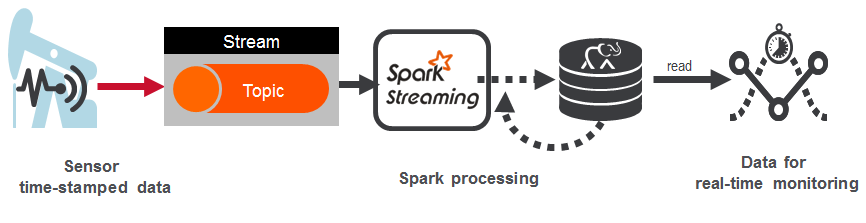


Example Use Case

Let's go over an example which generates lots of data and needs real-time preventive alerts. Remember what happened with BP in the Gulf coast?

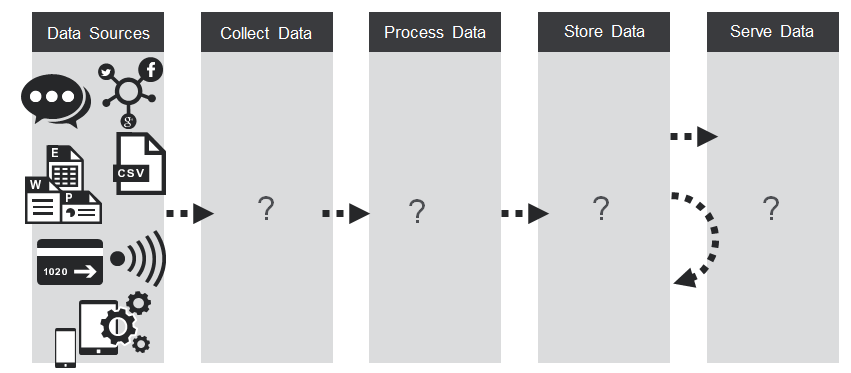


The example use case we will look at here is an application that monitors oil wells. Sensors in oil rigs generate streaming data, which is processed by Spark and stored in HBase, for use by various analytical and reporting tools. We want to store every single event in HBase as it streams in. We also want to filter for, and store alarms. Daily Spark processing will store aggregated summary statistics.



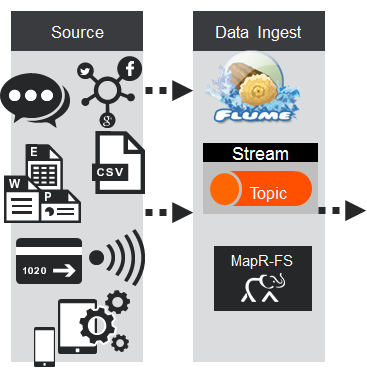
What do we need to do? And how do we do this with high performance at scale?

We need to collect the data, process the data, store the data, and finally serve the data for analysis, machine learning, and dashboards.

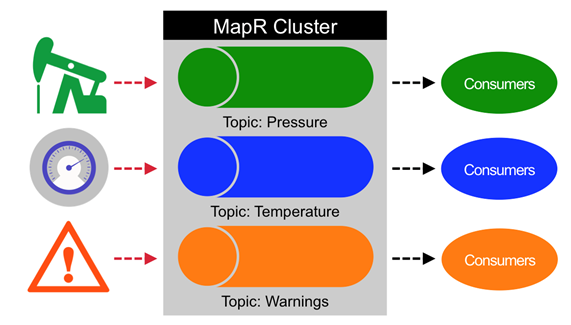


Streaming Data Ingestion

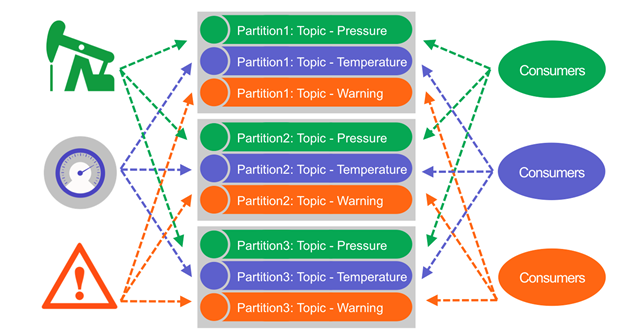
Spark Streaming supports data sources such as HDFS directories, TCP sockets, Kafka, Flume, Twitter, etc. In our example, we will use MapR Streams, a new distributed messaging system for streaming event data at scale. MapR Streams enables producers and consumers to exchange events in real time via the Apache Kafka 0.9 API. MapR Streams integrates with Spark Streaming via the Kafka direct approach.



MapR Streams (or Kafka) topics are logical collections of messages. Topics organize events into categories. Topics decouple producers, which are the sources of data, from consumers, which are the applications that process, analyze, and share data.



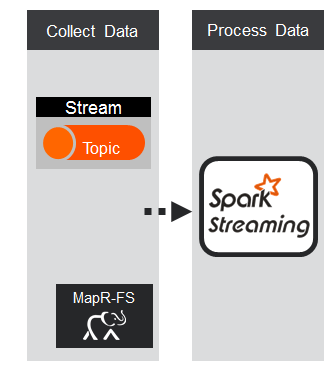
Topics are partitioned for throughput and scalability. Partitions make topics scalable by spreading the load for a topic across multiple servers. Producers are load balanced between partitions and consumers can be grouped to read in parallel from multiple partitions within a topic for faster performance. Partitioned parallel messaging is a key to high performance at scale.



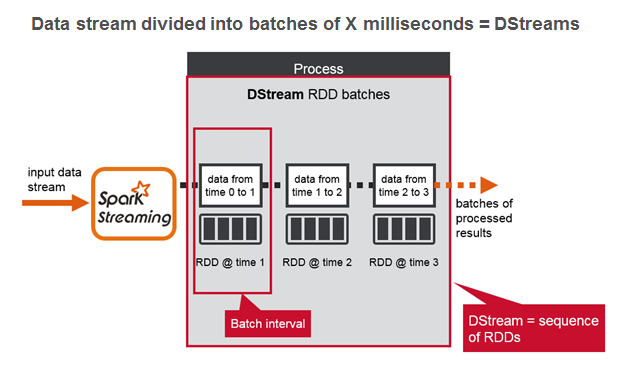
Another key to high performance at scale is minimizing time spent on Disk reads and writes.[Compared with older messaging systems, Kafka and MapR Streams eliminated the need to track message acknowledgements on a per-message, per-listener basis. Messages are persisted sequentially as produced, and read sequentially when consumed. These design decisions mean that non sequential reading or writing is rare, and allow messages to be handled at very high speeds.](https://www.mapr.com/streaming-architecture-using-apache-kafka-mapr-streams) MapR Streams performance scales linearly as servers are added within a cluster, with each server handling more than 1 million messages per second.

Real-time Data Processing Using Spark Streaming

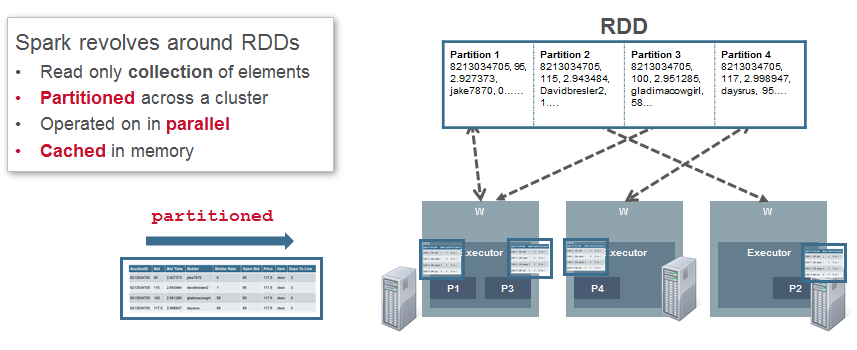
Spark Streaming brings Spark's APIs to stream processing, letting you use the same APIs for streaming and batch processing. Data streams can be processed with Spark’s core APIs, DataFrames, GraphX, or machine learning APIs, and can be persisted to a file system, HDFS, MapR-FS, MapR-DB, HBase, or any data source offering a Hadoop OutputFormat or Spark connector.



Spark Streaming divides the data stream into batches of X seconds called Dstreams, which internally is a sequence of RDDs, one for each batch interval. Each RDD contains the records received during the batch interval.

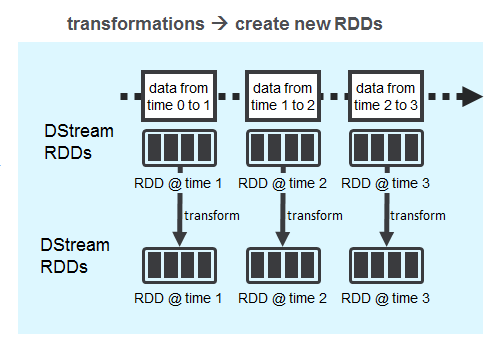


Resilient distributed datasets, or RDDs, are the primary abstraction in Spark. An RDD is a distributed collection of elements, like a Java Collection, except that it’s spread out across multiple nodes in the cluster. The data contained in RDDs is partitioned and operations are performed in parallel on the data cached in memory. [Spark caches RDDs in memory, whereas MapReduce involves more reading and writing from disk.](https://www.mapr.com/blog/parallel-and-iterative-processing-machine-learning-recommendations-spark) Here again the key to high performance at scale is partitioning and minimizing disk I/O.



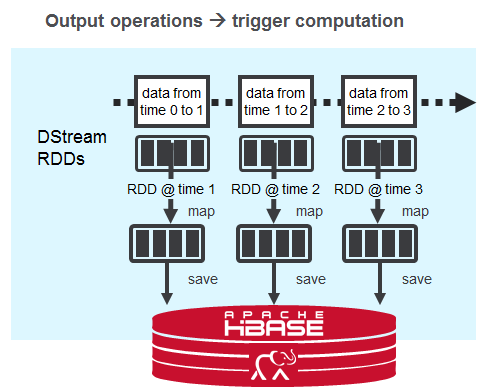
There are two types of operations on DStreams: transformations and output operations.

Your Spark application processes the DStream RDDs using Spark transformations like map, reduce, and join, which create new RDDs. Any operation applied on a DStream translates to operations on the underlying RDDs, which in turn, applies the transformation to the elements of the RDD.



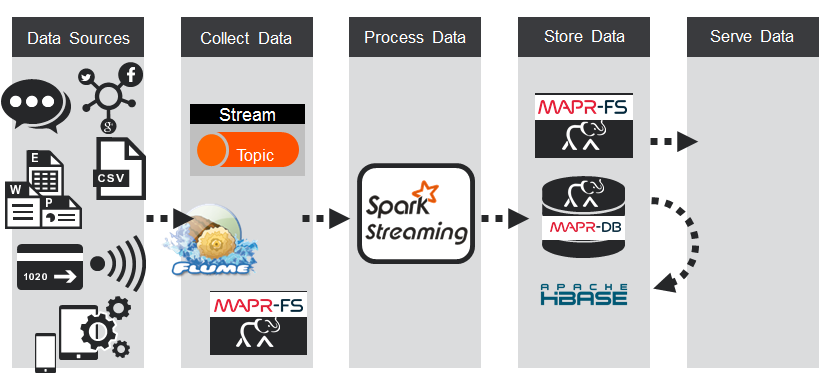
Output operations write data to an external system, producing output in batches.

Examples of output operations are saveAsHadoopFiles, which saves to a Hadoop-compatible file system, and saveAsHadoopDataset, which saves to any Hadoop-supported storage system.

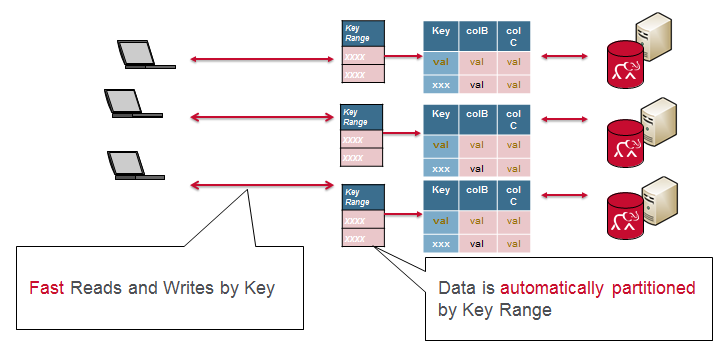


Storing Streaming Data Using HBase

For storing lots of streaming data, we need a data store that supports fast writes and scales.

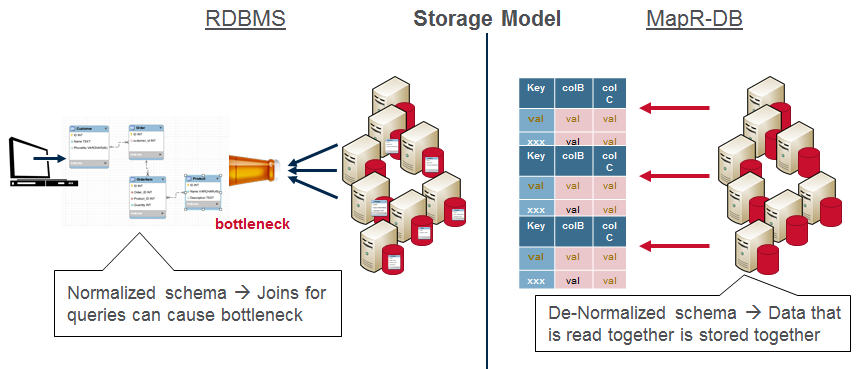


With MapR-DB (HBase API), a table is automatically partitioned across a cluster by key range, and each server is the source for a subset of a table. Grouping the data by key range provides for really fast read and writes by row key.



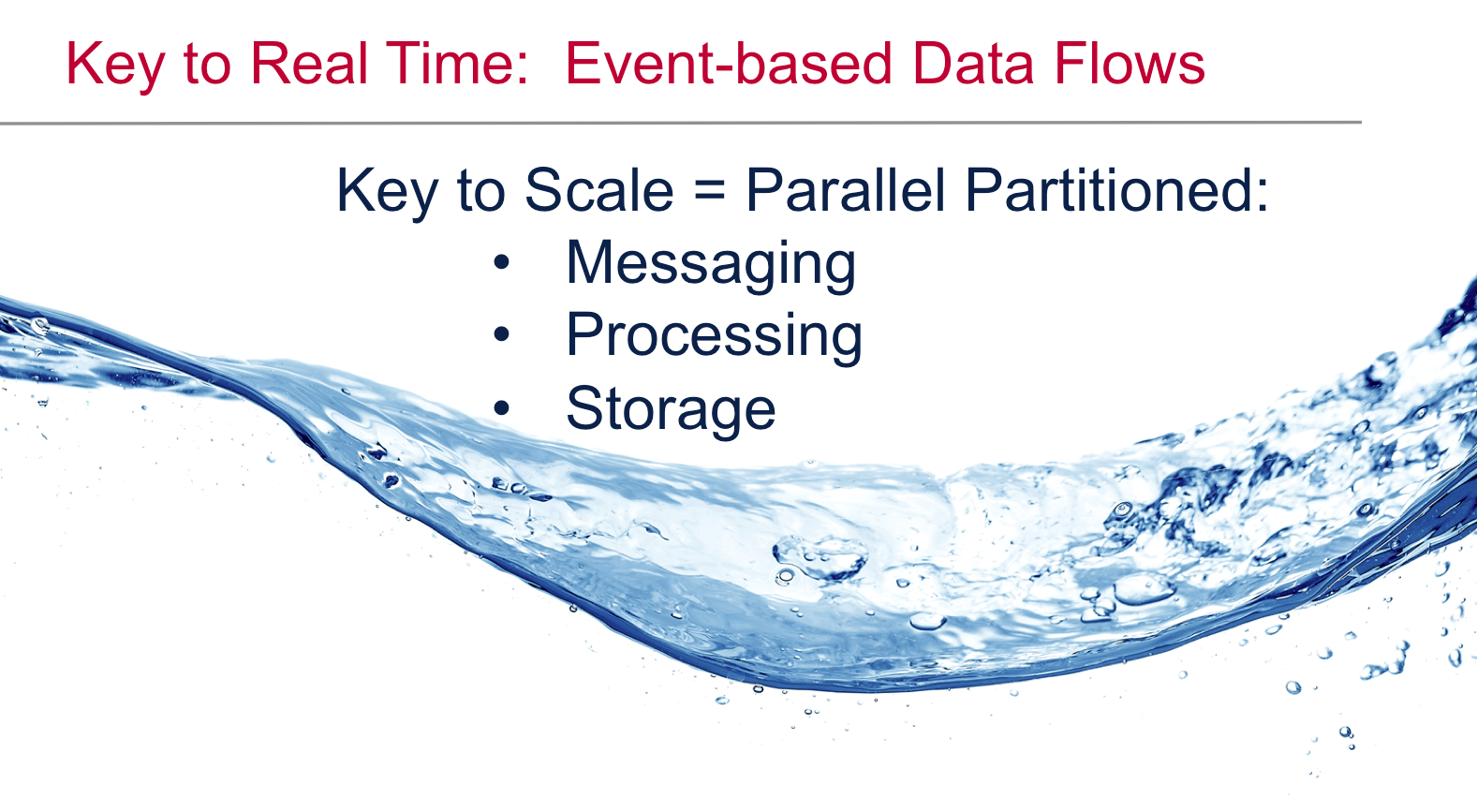
[Also with MapR-DB each partitioned subset or region of a table has a write and read cache. Writes are sorted in cache, and appended to a WAL; writes and reads to disk are always sequential; recently read or written data and cached column families are available in memory; all of this provides for really fast read and writes.](https://www.mapr.com/blog/in-depth-look-hbase-architecture)

With a relational database and a normalized schema, query joins cause bottlenecks with lots of data. MapR-DB and a de-normalized schema scales because data that is read together is stored together.



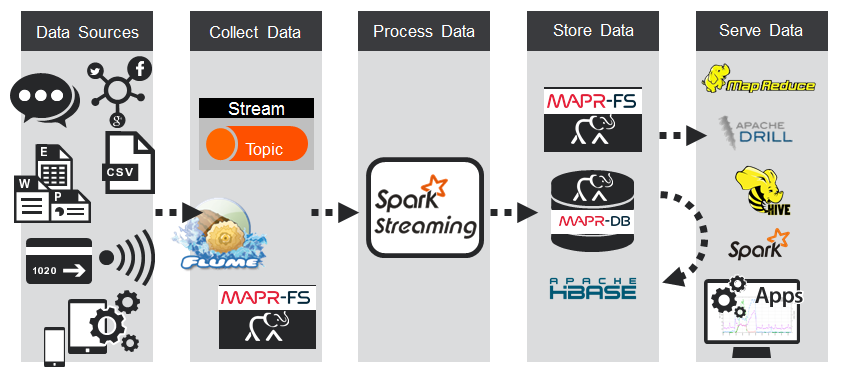
So how do we collect, process, and store real-time events with high performance at scale? The key is partitioning, caching, and minimizing time spent on Disk reads and writes for :

* Messaging with MapR Streams
* Processing with Spark Streaming
* Storage with MapR-DB



Serving the Data

End applications like dashboards, business intelligence tools, and other applications use the processed event data. The processing output can also be stored back in MapR-DB, in another Column Family or Table, for further processing later.



Example Use Case Code

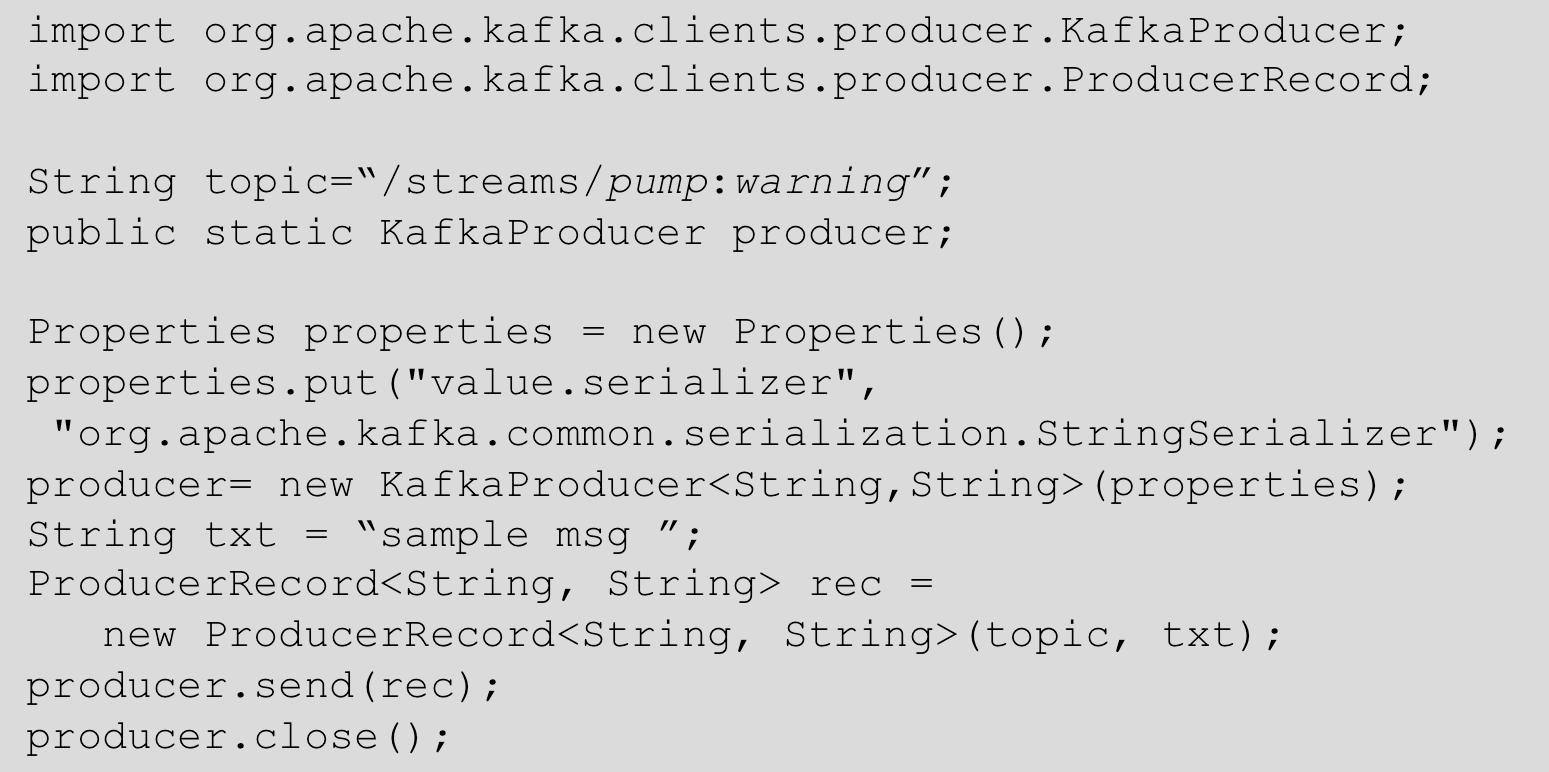
Now we will step through the code for a MapR Streams producer sending messages, and for Spark Streaming processing the events and storing data in MapR-DB.

MapR Streams Producer Code

The steps for a producer sending messages are:

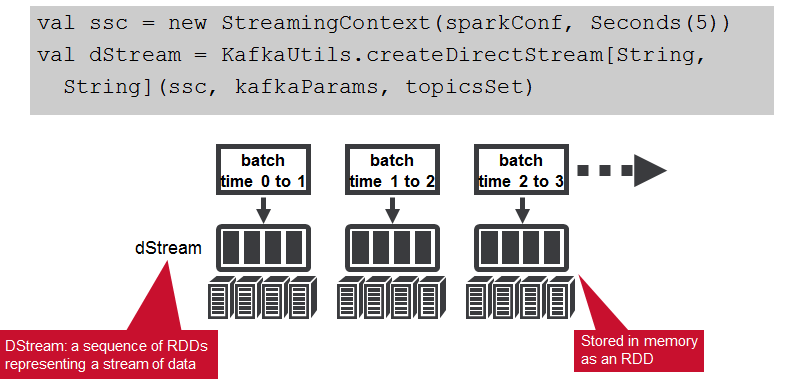
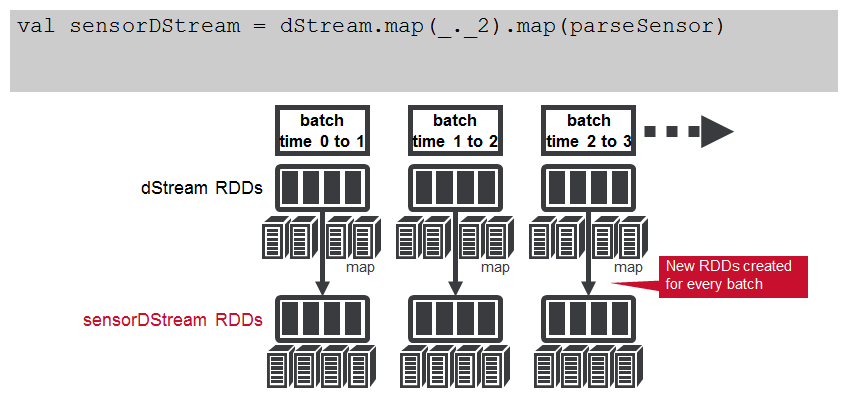
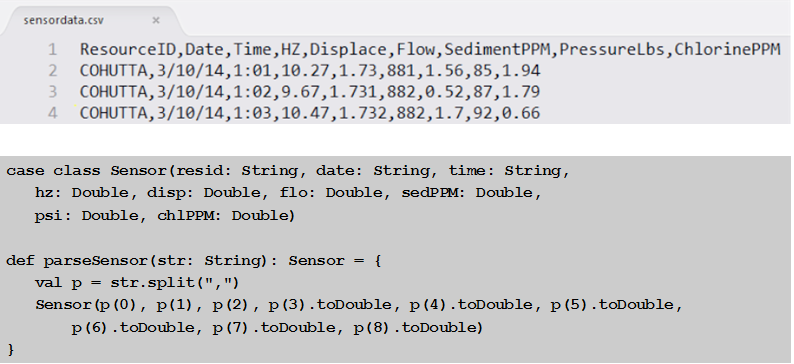
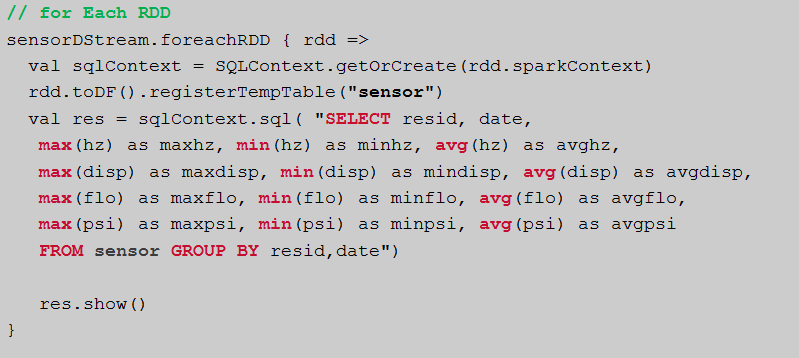
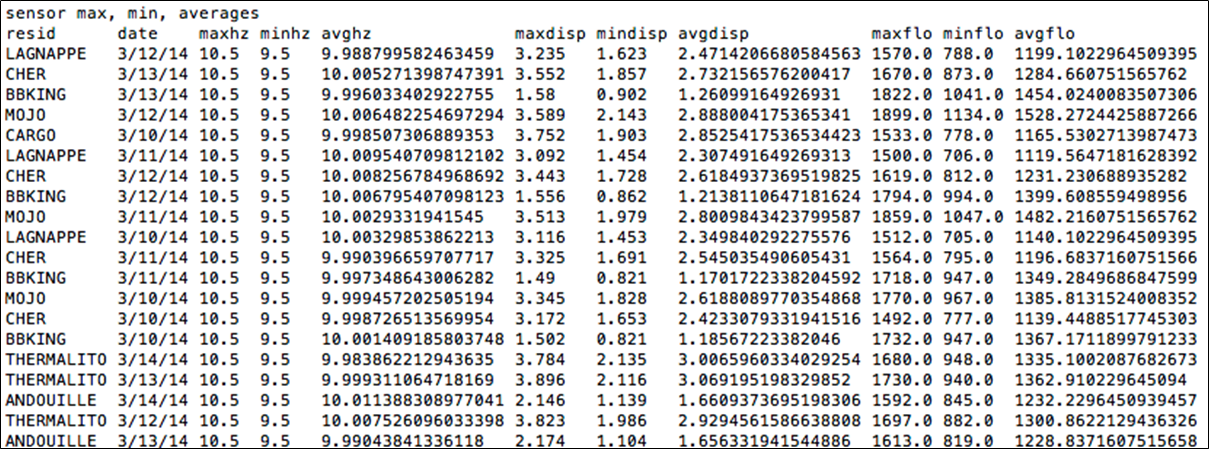
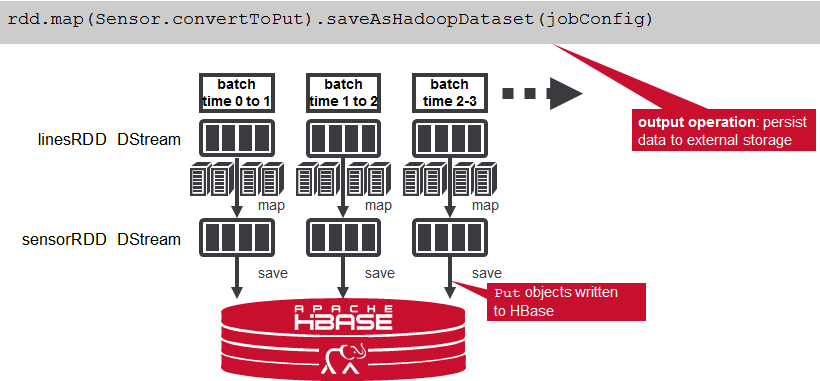
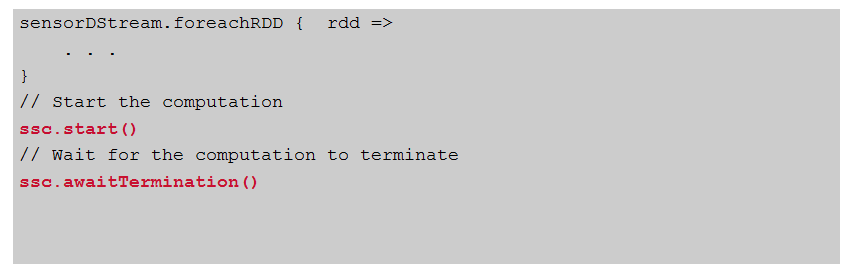
1. Set producer properties
   * The first step is to set the KafkaProducer configuration properties, which will be used later to instantiate a KafkaProducer for publishing messages to topics.
2. Create a KafkaProducer
   * You instantiate a KafkaProducer by providing the set of key-value pair configuration properties which you set up in the first step. Note that the KafkaProducer<K,V> is a Java generic class. You need to specify the type parameters as the type of the key-value of the messages that the producer will send.
3. Build the ProducerRecord message
   * The ProducerRecord is a key-value pair to be sent to Kafka. It consists of a topic name to which the record is being sent, an optional partition number, and an optional key and a message value. The ProducerRecord is also a Java generic class, whose type parameters should match the serialization properties set before. In this example, we instantiate the ProducerRecord with a topic name and message text as the value, which will create a record with no key.
4. Send the message
   * Call the send method on the KafkaProducer passing the ProducerRecord, which will asynchronously send a record to the specified topic. This method returns a Java Future object, which will eventually contain the response information. The asynchronous send() method adds the record to a buffer of pending records to send, and immediately returns. This allows sending records in parallel without waiting for the responses, and allows the records to be batched for efficiency.
5. Finally, call the close method on the producer to release resources. This method blocks until all requests are complete.

The code is shown below:



Spark Streaming Code

These are the basic steps for Spark Streaming code:

1. Initialize a Spark StreamingContext object. Using this context, create a DStream.
   * We use the KafkaUtils createDirectStream method to create an input stream from a Kafka or MapR Streams topic. This creates a DStream that represents the stream of incoming data, where each record is a line of text.
   * 
2. Apply transformations (which create new DStreams)
   * We parse the message values into Sensor objects, with the map operation on the dStream. The map operation applies the Sensor.parseSensor function on the RDDs in the dStream, resulting in RDDs of Sensor objects.
   * Any operation applied on a DStream translates to operations on the underlying RDDs. The map operation is applied on each RDD in the dStream to generate the sensorDStream RDDs.
   * 
   * The oil pump sensor data comes in as strings of comma separated values. We use a Scala case class to define the Sensor schema corresponding to the sensor data, and a parseSensor function to parse the comma separated values into the sensor case class.
   * 
   * Next, we use the DStream foreachRDD method to apply processing to each RDD in this DStream. We register the DataFrame as a table, which allows us to use it in subsequent SQL statements. We use an SQL query to find the max, min, and average for the sensor attributes.
   * 
   * Here is example output from the query which shows the max, min, and average output from our sensors.
   * 
3. And/or Apply output operations
   * The sensorRDD objects are filtered for low psi , the sensor and alert data is converted to Put objects, and then written to HBase, using the saveAsHadoopDataset method. This outputs the RDD to any Hadoop-supported storage system using a Hadoop Configuration object for that storage system.
   * 
4. Start receiving data and processing it. Wait for the processing to be stopped.
   * To start receiving data, we must explicitly call start() on the StreamingContext, then call awaitTermination to wait for the streaming computation to finish.
   * 

HBase Table schema

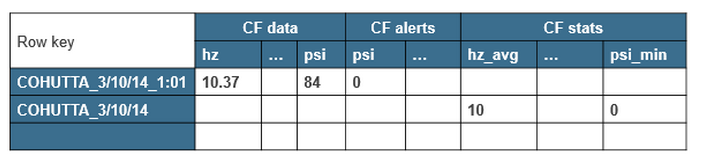
The HBase Table Schema for the streaming data is as follows:

* Composite row key of the pump name date and time stamp

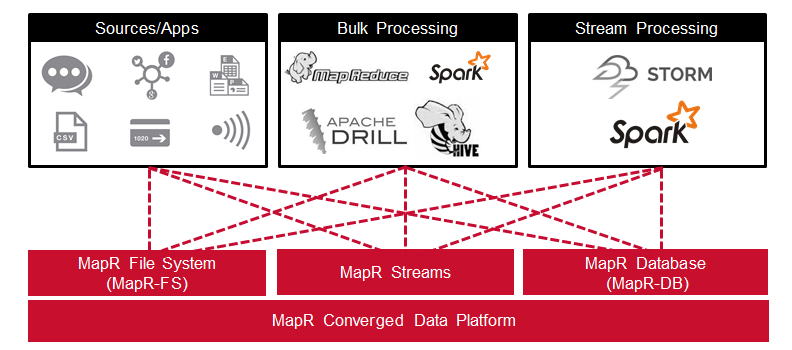
Column Family data with columns corresponding to the input data fields Column Family alerts with columns corresponding to any filters for alarming values Note that the data and alert column families could be set to expire values after a certain amount of time.

The Schema for the daily statistics summary rollups is as follows:

* Composite row key of the pump name and date
* Column Family stats
* Columns for min, max, avg.



All of the components of the use case architecture we just discussed can run on the same cluster with the MapR Converged Data Platform. There are several advantages of having MapR Streams on the same cluster as all the other components. For example, maintaining only one cluster means less infrastructure to provision, manage, and monitor. Likewise, having producers and consumers on the same cluster means fewer delays related to copying and moving data between clusters, and between applications.



Software

This tutorial will run on the MapR v5.1 Sandbox, which includes MapR Streams, Spark, and HBase (MapR-DB).

You can download the code, data, and instructions to run this example from here:

Code: <https://github.com/caroljmcdonald/mapr-streams-sparkstreaming-hbase>

Summary

In this blog post, you learned how the MapR Converged Data Platform integrates Hadoop and Spark with real-time database capabilities, global event streaming, and scalable enterprise storage.

References and More Information:

* [Free Online training on MapR Streams, Spark, and HBase at learn.mapr.com](http://learn.mapr.com/)
* [Getting Started with MapR Streams Blog](https://www.mapr.com/blog/getting-started-sample-programs-mapr-streams)
* [Getting Started with Apache Spark: From Inception to Production E-Book](https://www.mapr.com/getting-started-apache-spark?source=Social&campaign=2015_Social_Blog)