IoT & the Pervasive Nature of Fast Data & Apache Spark

Stephen Dillon
Schneider Electric, Global Solutions
stephen.dillon@schneider-electric.com

Last Modified 05/10/16

Table of Contents

Abstract

"Fast data" is not a new concept, although it is a relatively new term that is capturing the market's attention. Fast data is not just a buzzword. *It is a paradigm that defines the scope and activities within your IoT solutions needed to derive immediately actionable insights*. Companies are now embracing Fast Data as they seek to understand how to capitalize on their ROI in the Big Data and IoT domains.

The acceptance of The Internet of Things, by the mainstream, has served as a catalyst for Fast Data due to the demands and emphasis IoT places on the performance of *data ingestion, data processing, and actionable insights from your data*. There are many technologies that seek to support *at least some portion* of the needs of the Fast data paradigm and Apache Spark is one such technology that is proving itself able to address these needs for many use cases via its distributed compute and in-memory capabilities coupled with its streaming, SQL, Graph and Machine learning libraries.

This paper will introduce the Fast Data paradigm and provide a context within the scope of the Internet of Things and analytics. We will review Big Data and the architectural building blocks of Fast Data and then briefly survey the state of the art solutions in the open-source market whereas these are readily available to everyone regardless of budget constraints. We will then dive into Apache Spark as well as explore the Lambda architecture which is a popular approach to Fast Data and one Apache Spark supports well. We will conclude with a look towards what is next for Fast Data as the IoT market trends towards the need to support "Fog computing" a.k.a. Edge Computing use cases.

Introduction

"Fast Data" [1] has recently become one of the leading narratives in the Big Data and Analytics communities. This is largely due to the Internet of Things, which has served as a catalyst for its adoption, as companies try to best understand ways to realize their ROI in the quickly evolving market. However, Fast Data is not a new concept. *Data was fast before it was big, as the saying goes,* and we as experienced data professionals have spent decades seeking to address Fast Data under the guise of "performance"; whether via scale up database approaches, optimized database query engines, single-node partitioning strategies in RDBMS, or data warehousing solutions; and quite often unsuccessfully.

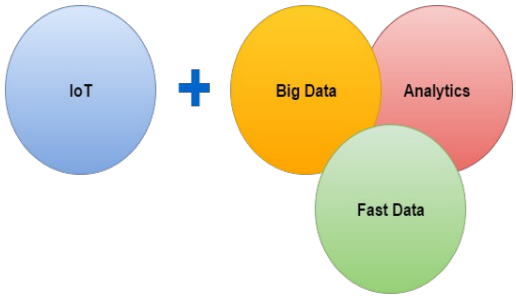

Illustration 1: Fast Data Domain Integration

Fast Data has become more relevant today due to the integration of IoT, Big Data and Analytics although it is not only limited to time-series or event data common to IoT sensors. It is applicable to all domains including finance, gaming, and astronomical studies and has evolved beyond just faster queries. It has become paramount in the quest to serve low-latency data processing, insightful decision making, and analytical computations and delivery leading to Smart Data for an evolving IoT market.

The concepts pertaining to Fast Data are often misunderstood, by the market at large, and warrant special attention *not only by data engineering professionals but by everyone working in the IoT domain*. The inspiration to write about this topic was greatly influenced by previous work I've done in the in-memory database space between 2012 and 2014 and my current work with Apache Spark. In those early days of discussing Fast Data a.k.a. "real-time" solutions it became quite apparent the market did not fully grasp where Big Data and IoT were trending. It was quite frustrating to hear people, back then, claim Fast Data was a solution *looking for a problem* yet many of these same folks are now very interested in Fast Data today for the same use cases. *I attribute this first to the* market's inability to see the application to existing use cases and second due to the lack of a mature open-source solution at that time. As such; the market is now trying to catch up to those few who did embrace Fast Data with commercial solutions. I hope that as you read this paper and are introduced to the concepts and technologies you will appreciate and understand the role of Fast Data and discover usefulness for your own projects.

IoT

Overview

IoT needs Fast data. It breeds it. It consumes it. IoT and the associated analytics foster the very existence of Fast Data *in the purview of the mainstream*. The nascent and ubiquitous data generated by IoT has been a catalyst for the adoption of Fast Data and thus we must understand its characteristics and practices and subsequently the supporting technologies. However; we must first fully understand what IoT is, to best understand why Fast Data is so important to IoT use cases, whereas there is a cause and effect for Fast Data's rise in popularity.

The common narrative for IoT is “A network of items—each embedded with sensors—which are connected to the Internet.” [2] It cannot be disputed that without these sensors “things” would not be able to do much with the Internet. *We also must acknowledge that we are not only referring to the actual sensors but also the data and analytics fueled by the sensor data.* This data can be generated and ingested at many tens of thousands of events per second and the ability to process this data, perform analytics, and gleam actionable insights quickly and efficiently is where the money is made and benefits are reaped in IoT. *Analytics is where the insights and business of IoT takes place.*

Influences of Fast Data

Fast Data has gained in popularity as c*ompanies realize that in order to derive insights from the data they must be able to react to it faster than batch analytics and Map Reduce approaches allow.* "Depending on use types, the speed at which organizations can convert data into insight and then to action is considered just as critical as the ability to leverage big data, if not more so. In fact, more than half (54%) of respondents stated that they consider leveraging fast data to be more important than leveraging big data. " [3]

Considering the disparate types of "things", i.e. sensors, and the forms the data has as well as the amount of it is not difficult to understand that IoT operates across all the three axis of Big Data i.e. Volume, Velocity, and Variety which in turn support analytics. NOSQL databases were quite successful at resolving the processing of both the "Variety" of multi-structured data from disparate devices as well as the "Volume" of data via the implementation of horizontally scaled out distributed data stores. These systems and frameworks such as Map Reduce further addressed the Velocity of data comparably *better than their predecessors in the RDBMS domain. This is not enough however* circa 2016. IoT solutions are now trending towards a need for more efficient and real-time a.k.a. "live" and interactive-analytics that are paramount to gain actionable insights from the type of data IoT generates. "By 2020 the digital universe – the data we create and copy annually – will reach 44 zettabytes, or 44 trillion gigabytes. " [4] *In order to process this data and derive actionable insights we must embrace Fast Data and do so in a secure and reliable (fault-tolerant) way.*

Big Data

Review

Let's take a step backwards to *briefly* review Big Data, whereas it will provide context for us to understand the Fast Data paradigm. *We should all be familiar* with the classic definition of Big Data i.e. "...data that exceeds the processing capacity of conventional database systems. The data is too big, moves too fast, or doesn't fit the structures of your database architectures. To gain value from this data, you must choose an alternative way to process it. ...The value of big data to an organization falls into two categories: analytical use, and enabling new products." [5] This definition led to the renowned three Vs of Big Data; Volume, Velocity, and Variety; yet many people mistakenly only associate Big Data with “Volume”. To be clear, *one need not possess all three Vs to have Big Data.* Any one of these axis can contribute to the inability of traditional systems to support the requirements *and thus you have Big Data* and the need to utilize the associated NOSQL and NewSQL technologies.

Significant Concepts

There are three significant concepts that came from the Big Data movement that are important to the Fast Data paradigm.

1. Distributed data storage
2. Horizontal, shared nothing, scale-out architectures
3. In-memory processing a.k.a. RAM is the new disk

With the exception of some graph databases, the myriad Big Data technologies and frameworks such as Hadoop, and the NOSQL and NewSQL databases, almost all utilized **distributed data and computing** based on a **horizontal shared-nothing architecture**. This allowed for data and computation to be dispersed among many commodity machines in a cluster and reduced costs compared to their scale-up counterparts in the RDBMS world.

NOSQL

The strength of NOSQL data stores is how they address Volume, via distributing data using a shared-nothing scale-out architecture, and Variety due to the lack of adherence to a strict schema. These design characteristics and their open-source origins mean we can implement solutions to address Big Data at a *significantly reduced time, effort and cost.* Each of these different types of NOSQL stores i.e. document, key-value, and column family, allow for varying degrees of support for Velocity and admittedly at orders of magnitude better than their predecessors in the RDBMS market. However, when compared to their counterparts in the NewSQL domain Velocity is not the strength of NOSQL and these technologies typically serve the needs of data scientists running offline batch analytics via the Map Reduce framework *and cannot solve the needs of Fast Data by themselves.*

NewSQL

NewSQL databases address the Velocity and Volume axis. They are typically distributed row or column stores that support at least a subset of the SQL standard and can serve OLTP or OLAP workloads. Like their NOSQL counterparts, they also utilize distributed computing and scale out architectures providing orders of magnitude better performance and scale than their predecessors in the RDBMS domain. One particular breed of NewSQL data store is the In-Memory Database System (IMDBS). IMDBS are based upon an in-memory first approach, **Illustration 2**, that first stores data in RAM and subsequently on disk in a fault-tolerant and reliable way. Data processing is generally performed against data in RAM in “real-time” which aligns them more closely with Fast Data. NewSQL databases are however comparatively less mature than NOSQL solutions when processing multi-structured data i.e. Variety. Pending your use case, such data stores may support your Fast Data needs via streaming and real-time analytics.

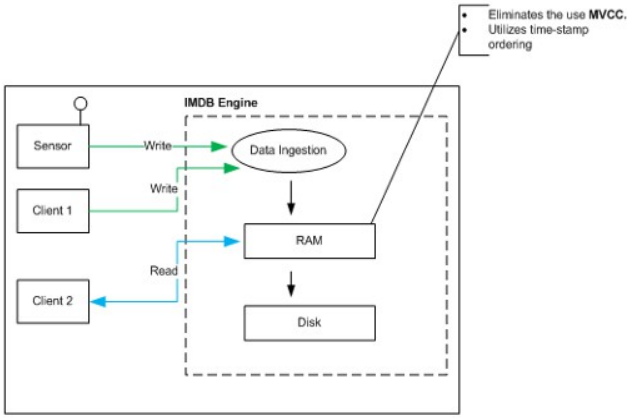

Illustration 2: IMDB Read\Write

Genesis of Fast Data

These underlying architectural concepts of NOSQL and NewSQL technologies served as the genesis of the modern Fast Data paradigm. Without them we certainly could not address the design concerns for low-latency computing and data storage. However, when coupled with these practices, data processing capabilities such as streaming, Graph analytics, and machine learning set the stage for rapid data ingestion, real-time decisioning, interactive analytics, and immediate results delivery which are the building blocks of Fast Data. Individually many of the associated NOSQL and NewSQL technologies such as MongoDB or VoltDB often play a role in meeting these demands via the Lambda Architecture which we will discuss in this paper. *Interestingly, the architectural principles of distributed computing, shared-nothing scale-out*

architecture, and the efficient use of RAM are key design considerations for Apache Spark.

Fast Data

Overview

Let's begin to define Fast Data by identifying it as *a subset of Big Data*. "Big data and Fast Data are the twin pillars of a leading-edge business strategy using applications and data. Although the twins perform different functions and address different challenges, both are key for companies wanting to transition to digital business." [6] It is often easiest for most people to understand Fast Data as associated with the Velocity axis however this does not *accurately portray* the story, scope or purpose and leaves us with a bit of confusion. This is akin to believing the Volume axis is the only characteristic of Big Data; which we know to not be true.

You may have heard the term "real-time" or "real-time processing" co-mingled in discussions of Velocity and people tend to apply this to data ingestion only. Although the term real-time accounts for a lot of what Fast Data encompasses this term was vague. Keep in mind that what is "real-time" for one use case is not considered "real-time" for use case. We must acknowledge Fast Data is more than simply receiving and ingesting a high frequency of data **although velocity is a key tenet.** "...it is not enough for companies to merely process data. Analyzing data to detect patterns, which can be immediately applied to maximizing operational efficiency, is the real driver of business value. " [7] This ability to detect patterns via complex analytics and machine learning is where I see the true power of Fast Data and in particular technologies such as Apache Spark.

Defining Fast Data

Fast Data truly encompasses the *Variety and Volume of data at* Velocity in all aspects. I prefer to view Fast Data as a paradigm that supports "...as-it-happens information enabling real-time decision-making." [8] which encompasses **not only the ingestion of data at speed** but also the processing of the data, deriving actionable insights from it, and the speed of delivery of the results. In layman's terms, we not only ingest the data quickly but we must process it, in all of its disparate formats, and perform our analytics on it immediately and often this is done against large volumes of data in-flight as well as at rest. *The Velocity at which you can ingest, process, perform analytics and act upon data with Variety and at Volume is what makes Fast Data unique.*

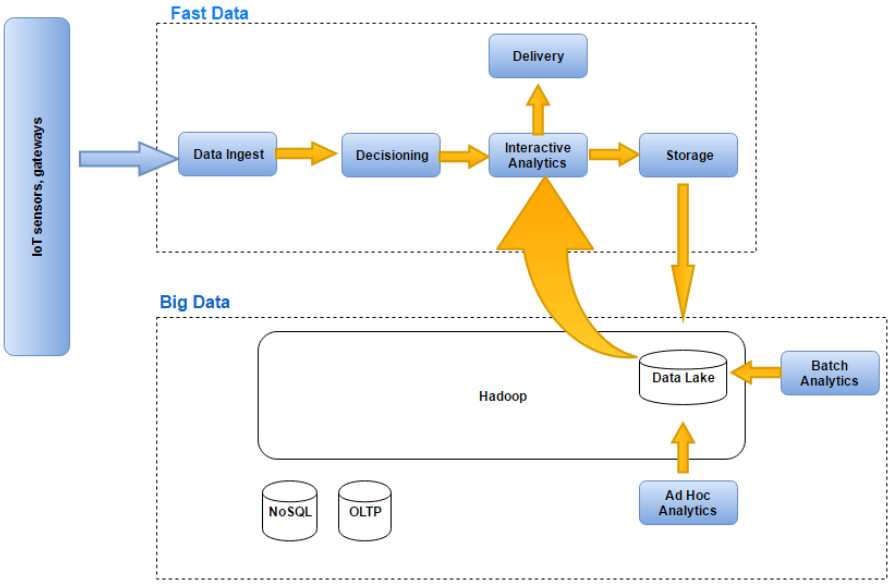

Illustration 3: Fast Data Workflow

Referring to Illustration 3, we can visualize a typical workflow for Fast Data that depicts the major activities that define Fast Data including **Ingest, Decisioning, Interactive Analytics, Storage, and Delivery**. Note that as we ingest data from our IoT devices, sensors, gateways et al, we do so *at Velocity and at the scale of up to millions of data points per second* and possibly from many disparate types of devices with different data points and formats i.e. multi-structured data. We then perform processing of this data, as it is streamed into our system, whether this be transformations or Lambda style

redirects of data to specialized targets asynchronously and in parallel. We then perform our interactive\real-time analytics against this data *in-memory* to support the low-latency requirements of IoT. An interesting aspect of this interactive analytics however is the joining of historical data, i.e. Data at rest, with our streaming "live" data. The historical data is that which we previously ingested perhaps hours, days, or years ago and have since stored in a Data Lake such as Hadoop or OLAP systems such as RedShift or Vertica to name a couple. We can additionally perform other non-interactive batch analytics against this data at rest whereas the analytics may be too complex or the data too large to be placed in RAM. In some examples the value of the analytics cannot be realized against the smaller amounts of streamed data alone and must be incorporated with historical data. Finally; once the analytics have been performed we must deliver the results in an efficient and timely manner. This is an often overlooked aspect of Fast Data and something that in-memory systems, such as Apache Spark or VoltDB, easily solve.

Concerns

If we take a step back and think about this within the context of not just Velocity but also Volume and Variety, we begin to realize a number of concerns such as how can we efficiently perform analytics on multi-structured data immediately? What if I have multiple streams of data for different kinds of IoT sensors being ingested at millions of data points per second and I wish to perform anlaytics against them in an integrated way? What if I wish to perform some deep graph analysis against these co-joined streams and additionally apply some historical batch data to the analytics? How many technologies will I need to implement and how can I integrate the results from multiple technologies and what overhead should I expect? Open-source technologies, such as Apache Spark, provide answers to such questions as well as ensuring reliability and cost reductions via a fault-tolerant and distributed design and the performance benefits of in-memory computing.

Lambda Architecture

Introduction

Fast Data does not adhere to any one particular architecture, although the Lambda Architecture (LA) is naturally suited to support IoT and Fast Data use cases. The LA allows one to incorporate streaming data, interactive analytics at speed, and historical batch analytics. I shall briefly review what the Lambda Architecture is whereas I have interviewed so-called "Senior architects" who have surprisingly not been familiar with it.

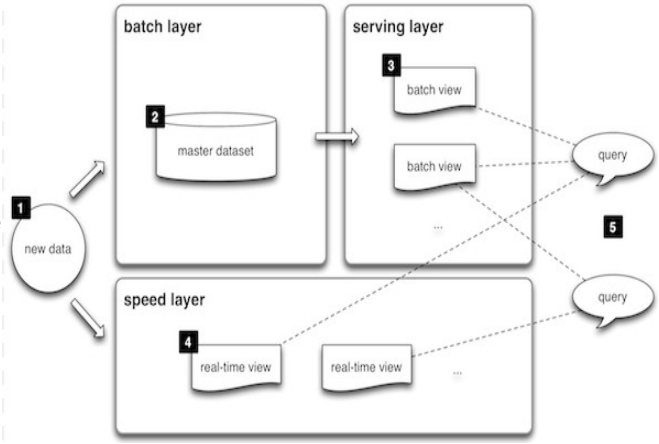

Illustration 4: [10]

The Lambda Architecture was an idea originally proposed by Nathan Marz and is described in detail in his book "Big Data , Principles and Best Practices". It is an approach to building stream processing systems on top of Map Reduce, Spark, and Storm or similar systems to process data asynchronously. "The LA aims to satisfy the needs for a robust system that is fault-tolerant, both against hardware failures and human mistakes, being able to serve a wide range of workloads and use cases, and in which low-latency reads and updates are required. The resulting system should be linearly scalable, and it should scale out rather than up. " [10] A key tenet of the LA is that it retains the original raw input data and it can replay the data if needed.

Layers

This architecture consists of multiple layers including a batch layer, a serving layer, and a speed layer as depicted in Illustration 4. The batch layer performs analytics on older data at rest. "The batch layer is usually a “data lake” system like Hadoop, though it could also be an OLAP data warehouse such as Vertica or Netezza. This historical archive is used to hold all of the data ever collected. The batch layer supports batch query; batch processing is used to generate analytics, either predefined or adhoc." [11]

The Speed layer however is "a combination of queuing, streaming and operational data stores. In the Lambda Architecture, the speed layer is similar to the batch layer in that it computes similar analytics - except that it computes those analytics in real-time on only the most recent data." [11]

Apache Spark supports the Lambda Architecture across both the batch and speed layers. Spark core and SparkSQL support the batch layer via its ability to query data at rest from HDFS, NOSQL or traditional databases. The Speed layer is supported by Spark's in-memory and Streaming capabilities. The data from each of these layers can be joined within Spark making it possible to perform analytics against both historical data and live data simultaneously including Graph analytics.

State of the Art

Overview

There are many Fast Data solutions on the market and many vendors have been active even prior to the emergence of the term "Fast Data". The commercial in-memory database vendors, such as the IMDB leaders VoltDB and MemSQL, have been touting in-memory as the key to highly efficient databases and have more recently begun using the term “Fast Data” to provide a more inclusive view beyond only real-time database operations. We've also begun to recently see some popular names in both the queuing and streaming domains such as Apache Kafka and Apache Storm also begin to take their place on the mantle of technologies participating in the Fast Data market albeit each of these only address pieces of the puzzle; especially when compared to Apache Spark . The one warning you should heed, when reviewing the market, is that despite a vendors' claims; there is no *one size fits all* solution. There are trade-offs to each solution and the key is to know your use case and to *fail fast* if needed.

It is albeit beyond the scope of this paper to cover the myriad offers in any great detail but we will review some of the more renowned *open-source* technologies in the next section. I purposely chose to only focus on the open-source solutions whereas they are readily available to everyone regardless of budget constraints although there are certainly other offers worthy of your time including VoltDB. Note the intention of the following review is not to provide a how-to or deep-dive into any one solution but instead to offer an introduction and enough information for one to contrast the capabilities of Apache Spark.

Apache Storm

Apache storm is an open-source distributed real-time computation system that can processes *unbounded* streams demonstrated in the range of millions of messages per second. Storm's architecture, see illustration 5, is based upon Spouts, which receive data from sources, and Bolts which perform some work and pass the results onto another Bolt or a target such as a data store or file system.

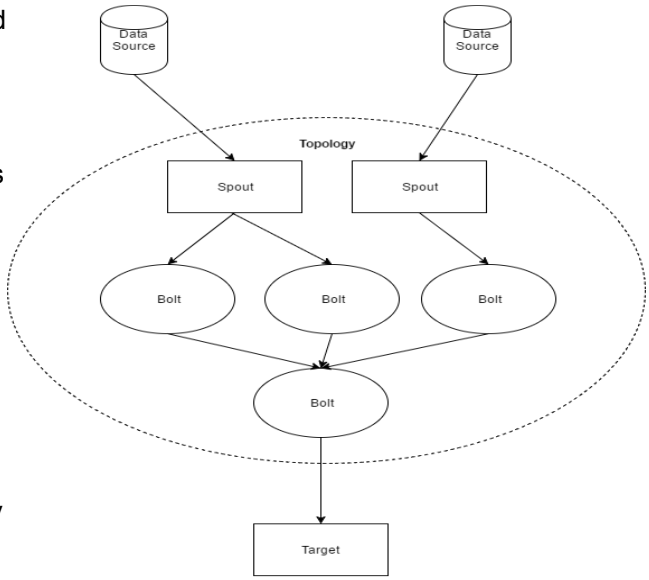

Illustration 5: Storm Topology

Many people experience a lot of confusion regarding the differences between Storm and Spark. Based on my experience, Storm supports a pure unbounded and continuous streaming model that allows one to integrate with queuing technologies like Kafka and most database systems. Unlike Spark, Storm does not provide any bundled graph or machine learning capabilities but instead allows you to create a topology that delivers data to other technologies that do fulfill these needs.

Storm is a true record by record streaming engine that supports the Lambda Architecture. It allows for one to write "bolts" that deliver data in parallel to different destinations (underlying databases or services). To date; if I had a use case focused on pure streaming *record by record* integrated with Hadoop then Storm would be a top contender.

Apache Flink

Flink is truly a Fast Data technology and the only true open-source *alternative to Apache Spark* as of the time of this paper. It is a scalable batch and stream processing platform that supports out-of-order events and exactly-once semantics. Flink provides very similar capabilities to Spark, as shown in Illustration 6, including Streaming, a Graph library, Machine learning library, and a Relational table library although at the time of this writing only streaming was known to be production ready. Flink purportedly does streaming “better” than Spark 1.6.1, due to its ability to support out of order data. This has triggered Spark to refine its own streaming strategy and improve performance as of Spark 2.0 circa May 2016.

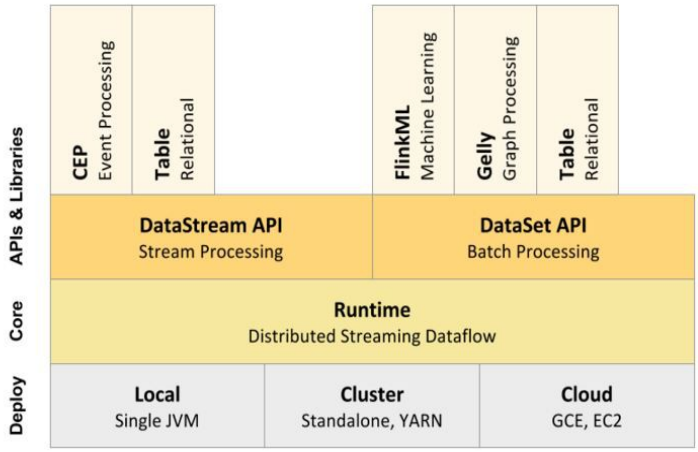

Illustration 6: Apache Flink Architecture [12]

One of the obstacles Flink is faced with is that it is not nearly as well known as Spark and the *majority* of the Flink user community is relegated to Germany. Spark has already had a number of years to mature and is more widely accepted by the market. It will be interesting to see how the market accepts Flink and if the major Cloud providers and Hadoop distributors integrate it as they have with Spark.

Apache Ignite

Apache Ignite is a memory data fabric which means it harnesses the memory of multiple nodes in a cluster to deliver data processing across data-sets in real time. In-memory computing is a key tenet of the Fast Data paradigm. Generally speaking, a memory fabric makes your chosen solutions faster. Distributed compute solutions such as Spark and Hadoop integrate nicely with Ignite making Spark *even faster than it already is.*

The Ignite stack, depicted in Illustration 7, goes beyond a mere memory fabric. It additionally provides numerous features such as a scalable and fault-tolerant continuous streaming capability and a distributed SQL key-value store that is ACID compliant allowing for distributed joins. Ignite is much more robust and serves a different, larger, purpose than Apache Storm or Spark whereas you can leverage it to integrate many other types of technologies. If you have solutions you wish to make faster, you can integrate them with Ignite potentially at the cost of a little more work on your part to perform the integration.

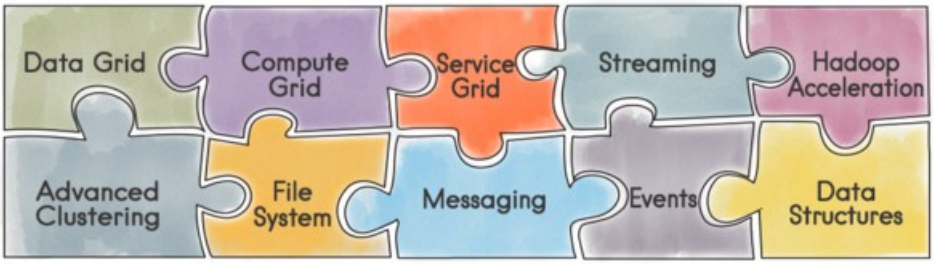

Illustration 7: Apache Ignite Stack [30]

Apache Spark

Overview

Apache Spark is a distributed compute engine that supports Fast Data via its in-memory and distributed processing capability as well as its bundled APIs for Streaming, SQL, MLLib, and Graph. It can "run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk." [13] Originally created at the Berkely AMPLab, it is commercially supported by Databricks, the company founded by its creators although Spark is fully open-source and available to all. It has become one of the fastest growing open-source solutions since 2014 and it has emerged as the defacto standard for data engineering and data science tasks.

Spark utilizes a Resilient Distributed Dataset (RDD) as its core data structure. An RDD is an immutable, distributed collection of objects and operations on RDDs are fault-tolerant. Spark can automatically reconstruct any data or replay tasks lost due to failure. “Each RDD is split into multiple partitions, which may be computed on different nodes of the cluster." [14] All other higher level data structures in Spark, i.e. Dataframes and Datsets, are based atop of RDDs and are applicable across the various APIs. I encourage you to refer to the Databricks online documentation or to the book "Learning Spark" by Kanau et al [15].

Misconceptions

Before we review the Spark packages, let's address a few misconceptions regarding what Spark is and does whereas I hear the following topics raised frequently.

1. *It is not a database.* If you are concerned about data storage and ACID compliance or disaster recovery and backups for manipulated data, you're underlying data stores should support these needs.
2. It **does not** compete with Hadoop. It is complimentary to it. They are both Big Data frameworks and serve different purposes.
3. It **does not** provide its own distributed file system. This is why we often see Spark implemented atop Hadoop due to HDFS although Spark can be used with other distributed file systems as well.
4. Spark does not perform every operation in-memory nor store all data in memory. In order to cache data in memory, one must explicitly enable it via the persist and cache functions on an RDD.

Packages

One of the most salient differences between Spark and the aforementioned state of the art technologies, with the exception of Flink, is the multiple components within the Spark ecosystem as depicted in **Illustration 8**. A Fast Data solution need not support each of these packages although Spark has seen it pertinent to bundle them together whereas they are commonly used by data scientists.

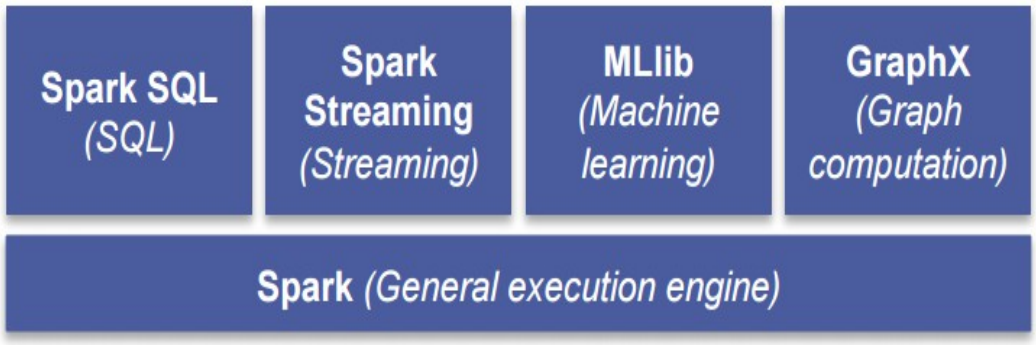

Illustration 8: Apache Spark Stack [9]

Spark eliminates the need to create a mixture of solutions to support real-time analytics and processing, machine learning, and graph requirements. Although each component performs a different function the similarity across each API makes it very easy for engineers to leverage each. Spark's ability to support a Lambda Architecture by implementing various contexts to stream data (in micro batches) in memory and then perform machine learning and Graph operations also makes Spark a very attractive solution.

Streaming

It is important to understand, in the world of Streaming technologies, that *Spark Streaming is not a continuous streaming or record-by-record solution.* It instead processes streams using a "micro-batch architecture, where the streaming computation is treated as a continuous series of batch computations on small batches of data." [15] The micro-batch architecture is one of the top complaints I have heard within the user community but in reality this affects very few use cases unless you are processing data within a window less than 500 milliseconds. In my experience it is reasonable to configure Streaming at intervals as low as 1/2 a second *pending your use case and the data workload*. You must however test your proposed streaming frequency per your use case. The key is to ensure the previous micro-batch can be processed, in full, before the next one is delivered.

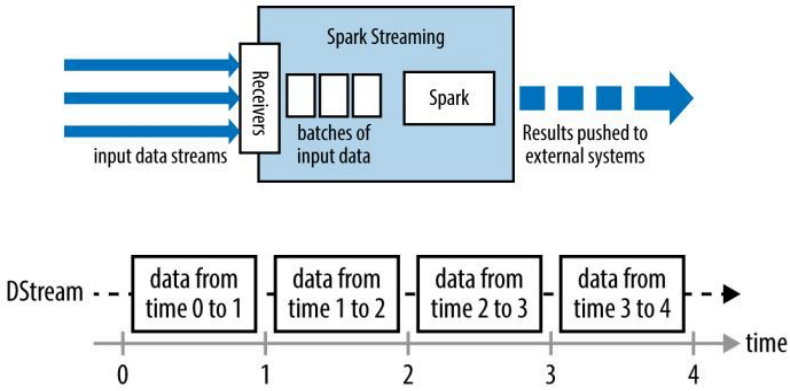

Illustration 9: Spark Streaming [22]

Spark's streaming API is very similar to the other Spark components and very easy to implement. The following illustration depicts a typical Word count sample running on my cluster written in Scala. A streaming context is passed to a function that instantiates a streamingContext. The streaming context is used to retrieve new files as they are added to HDFS or in this case Azure's "wasb" blob storage.

```
def runHDFSStreamer(sc: SparkContext): Unit = {
  val ssc = new StreamingContext(sc, Seconds(1))

  // Create the FileInputDStream on the directory and use the
  // stream to count words in new files created
  val lines = ssc.textFileStream("wasb://streamdata@sdddev1.blob.core.windows.net/")
  val words = lines.flatMap(_.split(" "))
  val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
  wordCounts.print()
  ssc.start()
  ssc.awaitTermination()
}
```

Illustration 10: Streaming Code Sample

Exactly once semantics

Spark Streaming supports exactly once semantics which is a practice of processing a stream's content *one time and thus avoiding duplicate data*. It is important to understand that the promise of exactly once processing is *only applicable to the input data*. "In order to achieve exactly-once semantics for output of your results, your output operation that saves the data to an external data store must be either idempotent, or an atomic transaction that saves results and offsets." [16]

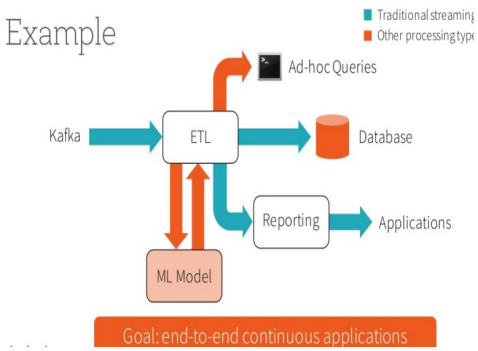

Illustration 11: Structured Streaming - Strata & Hadoop World April '16

"Due to Spark Streaming's worker fault-tolerance guarantees, it can provide exactly once semantics for all transformations-even if the worker fails and some data gets

reprocessed, the final transformed result...will be the same as if the data were processed exactly once." [17] Spark 2.0, due in May of 2016, will introduce a new Structured Streaming Engine that is purported to allow developers to do more with streams. This includes better support for data output, better windowing capabilities, and the ability to perform adhoc queries and machine learning algorithms against the stream.

SparkSQL

Apache Spark supports *a subset* of the Structured Query Language (SQL) for querying a variety of data sources including relational data stores, files of various formats on HDFS such as JSON or Parquet, as well as NOSQL data stores such as MongoDB, Hive, HBase and Cassandra via external libraries. Spark supports querying these data sources as tabular data structures by implementing a "SchemaRDD" interface. The SchmeaRDD is built either programmatically or through inferencing the schema of a regular RDD. This schemaRDD is then stored as a queryable temporary table.

A very interesting aspect to executing SQL against NOSQL data stores is the ability to push the actual SQL predicates i.e. WHERE clauses *down to the originating data source* such as MongoDB. One need not return more data than they need to the application for further filtering. For example, the following code sample depicts how one can query a MongoDB collection consisting of millions of IoT sensor data points without retrieving all of the collection's data.

```
def parseMongoSQL(sqlCntxt: org.apache.spark.sql.SQLContext) {

    val mcInputBuilder = MongodbConfigBuilder(Map(Host -> List("mongosparklx2:27017"),
    Database -> "timeseries", Collection -> "BOC-SBO-XVSFD-SDF", SamplingRatio -> 1.0,
    WriteConcern -> "normal"))

    val readConfig = mcInputBuilder.build()

    //this is passive and pulls no data until the SQL below is executed
    val rdd = sqlCntxt.fromMongoDB(readConfig)
    rdd.registerTempTable("timeseries")

    //the WHERE clause is pushed to MongoDB and the query is executed there (see MongoDB profiler)

     val res = sqlCntxt.sql("""SELECT DISTINCT """ +
                                    "timeSeriesId, " +
                                    "historyRecords.timestamp, " +
                                    "historyRecords.value " +
                                    "FROM timeseries " +
                                    "WHERE historyRecords.value > 45 and historyRecords.value < 72 " +
                                    "limit 10")


    //display the contents in the spark shell
    res.collect().foreach(println)
  }
```

Illustration 12: Querying MongoDB with SparkSQL

Machine Learning

Spark provides scalable machine learning capabilities using well-known algorithms such as classification, regression models, and clustering. If you are unfamiliar with Machine Learning, it is a practice of training a data model on some input data and making predictions. For example, in 2003 I wrote my own neural network that I trained to predict the outcomes of NFL games. I provided the network with the results of previous seasons' as a training set and after many iterations against a back propagation model I was able to determine the best inputs and weights to apply to the network to achieve the highest percentage of successful predictions for future games. ** I won the football pool that year.*

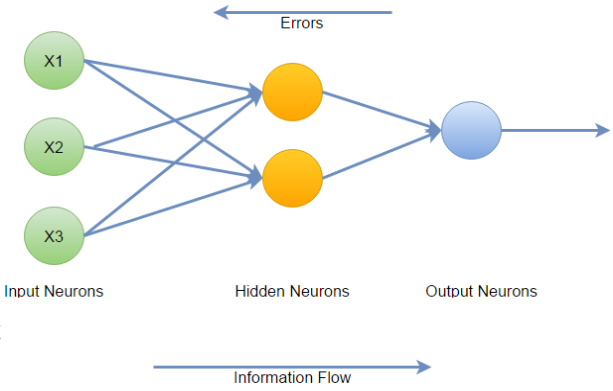

Illustration 13: Neural Network - Back Propagation

Now consider that in IoT we have millions of data points being streamed every second for robots on factory floors, generators, wind turbines, energy management devices, automobiles and airplanes. You can imagine the desire of companies to be able to predict equipment failure or battery life of critical components in remote areas. It is far beyond the scope of this paper to delve into the details and the variety of Machine Learning algorithms we could apply to these potential use cases. We are only concerned here with the fact Spark provides a machine learning library and allows us to apply these learning models across distributed data in parallel and join these operations with other data using the Spark APIs such as the Graph processing as discussed next. If you are interested in Machine Learning, I encourage you to refer to the Spark MLlib documentation [18] for a complete list of the available algorithms.

GraphX

GraphX is a distributed parallel processing graph engine that supports the processing of Graphs and subgraphs.

A graph is a data structure, represented in Graph theory as G = (V,E), consisting of Vertices a.k.a. nodes that have edges and properties. There are *generally two types of graphs that* we will come across. The first is a Property graph and the second is RDF (Resource Description Framework) graphs which are stored as "triples" in the form of subject, object, and predicate.

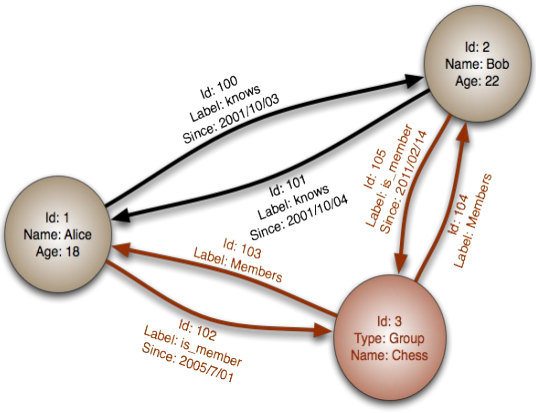

Illustration 14: Graph Relationships [28]

Why Graphs? Graphs enable you to analyze relationships as opposed to more traditional analytics that perform aggregations and report on data. "What can graph analytics accomplish that other analytic approaches cannot? Based on graph mathematical theory, graph analytics model the strength and direction of relationships within a given system. Graph analytics can be used not only to detect a correlation, but also to determine its nature and how significant it really is within the overall system." [19] Essentially graph analytics allow you to *find answers to questions when the questions are not known in advance* such as the relationships between items in a network.

"GraphX represents graphs internally using two Spark distributed collections (RDDs) – an edge collection and a vertex collection. By default, the edges are partitioned according to their configuration in the input collection (e.g., original placement on HDFS). However, they can be re-partitioned by their source and target vertex ids using a user-defined partition function." [20] Databricks has additionally introduced

GraphFrames, which is a graph processing library for Apache Spark that is built atop Spark Dataframes. GraphFrames provides the additional ability to work with Graph data using SparkSQL and to read and store graph data from formats including JSON, Parquet and CSV. [21]

Fast Data is Evolving

The Fast Data paradigm is not static. It is evolving and keeping pace with the evolution of IoT as the number of devices continue to grow and become more mobile via the influx of wearables and connected vehicles for example. The addition of mobility to sensors further inspires an already growing need for in-flight real-time processing and low-latency analytics in quite possibly not yet realized ways circa 2016. This will only further complicate matters as the IoT community is already experiencing bottlenecks in the Cloud due to the amounts of data collected and sent to a centralized system. In many scenarios it does not make sense to send data from many devices, located at a campus or on a user's wrist, via the Cloud to a *centralized system* to process data, perform analytics on it, and send a result back to the devices. It may instead be pertinent to process the data as near as possible to the devices although many of these devices simply do not have the memory, storage, or processing power to satisfy these needs.

As this need to reduce time to actionable insights becomes critical, one natural way to reduce latency is to push the data processing and computations nearer the devices that make up the Internet of Things. To facilitate this need, Fast Data integrates nicely with another paradigm that is finding its way into the consciousness of the mainstream; "Fog computing" a.k.a. Edge computing.

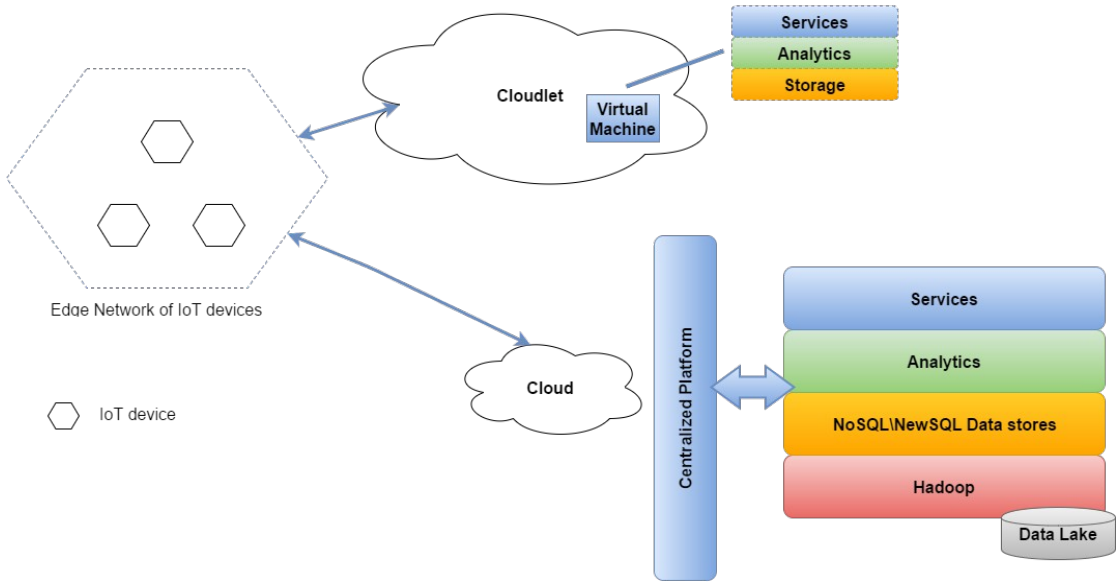

Illustration 15: Fog Computing Landscape

The architecture depicted in illustration 15 places the data processing and analytics capabilities in a cloudlet located near the device. A cloudlet is an architectural element that originated in mobile and cloud computing. “A cloudlet can be viewed as a "data center in a box" whose goal is to bring the cloud closer. A cloudlet is a trusted, resource-rich computer or cluster of computers that's well-connected to the Internet and available for use by nearby mobile devices.“[23] Rather than collect and send all data to a centralized system in the cloud, certain local parameters can be processed on the edge of the network of devices, in these cloudlets, thus reducing network latency and time to actionable insights. Data deemed worthy of deeper analysis and longer term storage will be transmitted to back-end systems in the Cloud.

Conclusion

The very nature of IoT and analytics is cyclical and the evolving demands for faster and more complex analytics will continue to serve as a catalyst for the adaptation of Fast Data and technologies such as Apache Spark. As IoT devices become more mobile and ubiquitous they will continue to change how we live and work and increase our demands for connectivity. These behavioral changes and the availability of the data will continue to create new demands for analytics and new ways to discover insights that are delivered and acted upon immediately. Therefore we must not only concern ourselves with the Velocity of the data being ingested but Velocity at all stages of the data life cycle to support these pervasive analytics.

Technologies such as Apache Spark are at the forefront of meeting these robust demands coupled with reliability and fault-tolerant distributed computing. As IoT architectures evolve to a Fog Computing paradigm, fast and flexible technologies such as Spark will continue to be paramount and must continue to evolve to support the purported tens of billions of devices that will be connected to our vehicles, homes, and wrists. Only by embracing the lessons of Fast Data and technologies such as Spark will we be able to support the IoT of tomorrow.

About the Author

Stephen Dillon is a Data Architect working at Schneider Electric on the Global Solutions team where he is focused on the corporate IoT Platform and Big Data technologies. He is a senior member of Schneider's Engineering Fellow program (EDISON Experts) and he holds a Masters Degree in Computer Information Systems from Boston University. He has over 17 years of experience focusing on database technologies and data architecture for platforms and enterprise systems. He has been working with energy management meters since 2009, Big Data since 2011 and contemporary Fast Data technologies since 2012 and programmed & connected his first IoT device circa 2003.

You may reach Stephen via email at stephen.dillon@schneider-electric.com, Linkedin at https://www.linkedin.com/in/stephendillon as well as Twitter @stephendillon15.

References

[1] Baer T., “What is Fast Data”, Ovum, Nov 2012

[2] Pretz K., Smarter Sensors, Making the Internet of Things Soar, March 14, 2014, available via http://theinstitute.ieee.org/technology-focus/technology-topic/smarter-sensors

[3] Cagemini, "Big & Fast Data: The Rise of Insight-Driven Business ", available via https://www.capgemini.com/resource-file-access/resource/pdf/big_fast_data_the_rise_of_insight-driven_business-report.pdf

[4] Dumbill E., “What is Big Data?”, July 11, 2012, available online via https://www.oreilly.com/ideas/what-is-big-data

[5] EMC, “The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things”, available online via http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm

[6] TIBCO, “Fast Data and Architecting the Digital Enterprise”, available online via http://www.tibco.com/assets/bltcc05dbd25dfe897a/wp-fast-data-and-architecting-the-digital-enterprise.pdf

[7] MemSQL, “The Lambda Architecture Simplified, MemSQL”, April 2016

[8] Alissa Lorentz, “Big Data, Fast Data, Smart Data”, available online via http://www.wired.com/insights/2013/04/big-data-fast-data-smart-data/

[9] Zaharia M., 2016 Spark Summit Keynote speech

[10] http://lambda-architecture.net/

[11] Piekos J., “Simplifying the (complex) Lambda Architecture”, December,1,2014, available online via https://voltdb.com/blog/simplifying-complex-lambda-architecture

[12] Apache Flink Internals, available online via https://ci.apache.org/projects/flink/flink-docs-release-1.0/internals/general_arch.html

[13] Apache Spark, http://spark.apache.org/

[14] Karau, Konwinski, Wendell, Zaharia, "Learning Spark, Lightning Fast Data Analysis", pp. 23, O'Reilly, 2015, ISBN-13: 978-1449358624

[15] Karau et al, pp. 188

[16] Apache Spark 1.6.1, "Spark Streaming + Kafka Integration Guide", available online via https://spark.apache.org/docs/1.6.1/streaming-kafka-integration.html

[17] Karau et al, pp. 211

[18] Spark, Machine Learning Library (MLLOB) Guide, available online via https://spark.apache.org/docs/1.6.1/mllib-guide.html

[19] Hoskins M., “How Graph Analytics Deliver Deeper Understanding”, available online via http://www.infoworld.com/article/2877489/big-data/how-graph-analytics-delivers-deeper-understanding-of-complex-data.html

[20] Reynold S. Xin et al, "GraphX: Unifying Data-Parallel and Graph-Parallel Analytics", Section 4.2 Distributed Graph Representation, UC Berkeley AMPLab

[21] Hortonworks, "In-memory Processing With Apache Spark", March 12th 2015, available online via

http://hortonworks.com/webinar/memory-processing-apache-spark/

[22] Bradley J. et al, "Databricks, Mastering Advanced Analytics with Apache Spark", Databricks

[23] Santayana M., Tha Case for VM-Based Cloudlets in Mobile Computing, available online via http://elijah.cs.cmu.edu/DOCS/satya-ieeepvc-cloudlets-2009.pdf

[28] Schmarzo, B., Graph Analytics 101, January 28, 2014, available online via https://infocus.emc.com/william_schmarzo/graph-analytics-101/

[30] Apache Ignite, https://ignite.apache.org/

[31] Koeninger C., Lie D., Das T., “Improvements to Kafka integration of Spark Streaming”, https://databricks.com/blog/2015/03/30/improvements-to-kafka-integration-of-spark-streaming.html