WHITE PAPER

Fast Data: Big Data Evolved

Dean Wampler, Ph.D.

Office of the CTO and Big Data Architect

Typesafe

Table of Contents

Fast Data: Big Data Evolved

Big Data got its start in the late 1990s to early 2000s when the largest Internet companies were forced to invent new ways to manage data of unprecedented volumes. Today, most people think of **Hadoop** or NoSQL databases when they think of Big Data. However, the original core components of Hadoop, ***HDFS*** (Hadoop Distributed File System - for storage), ***MapReduce*** (the compute engine), and the resource manager now called ***YARN*** (Yet Another Resource Negotiator) are rooted in the *batch-mode* or *offline* processing commonplace ten to twenty years ago, where data is captured to storage and then processed periodically with batch jobs. Most search engines worked this way in the beginning. The data gathered by web crawlers was periodically processed into updated search results.

At the other end of the processing spectrum is real-time event processing, where individual events are processed as soon as they arrive with tight time constraints, often microseconds to milliseconds. The guidance systems for rockets are an example, where behavior of the rocket is constantly measured and adjustments must be made very quickly.

Between these two extremes are more general stream processing models with less stringent responsiveness guarantees. A popular example is the *mini-batch* model, where data is captured in short time intervals and then processed as small batches, usually within time frames of seconds to minutes.

The importance of streaming has grown in recent years, even for data that doesn't strictly require it, because it's a competitive advantage to reduce the time gap between data arrival and information extraction.

For example, if you hear of a breaking news story and search Google and Bing for information, you want the search results to show the latest updates on news websites. So, batch-mode updates to search engines are no longer acceptable, but a delay of a few seconds to minutes is fine.

Stream processing is also being used for these tasks:

- Updating machine learning models as new information arrives.
- Detecting anomalies, faults, performance problems, etc. and taking timely action.
- Aggregating and processing data on arrival for downstream storage and analytics.

Even batch-mode processing is experiencing a renaissance. The performance of individual HDFS and MapReduce services wasn't a priority at the time, yet Hadoop delivered good performance over massive data sets. Like discount retailers, how did they do it? Volume! Partitioning data and processing the

partitions in parallel made up for local inefficiencies. However, with rising concerns about infrastructure costs, efficient batch computation is also desirable.

Finally, Big Data has become the *killer app* for functional programming (FP) and functional languages like Scala. The emphasis on immutability improves robustness, and data pipelines are naturally modeled and implemented using collections (like lists and maps) with composable operations. Object-oriented programming is a less useful approach, so functional languages are surging in popularity.

The phrase ***Fast Data*** captures this range of new systems and approaches, which balance various tradeoffs to deliver timely, cost-efficient data processing, as well as higher developer productivity. Let's begin by discussing an emerging architecture for Fast Data.

A Fast Data Architecture

What high-level requirements must a Fast Data architecture satisfy? They form a *triad*:

1. Reliable data ingestion.
2. Flexible storage and query options.
3. Sophisticated analytics tools.

The components that meet these requirements must also be ***Reactive***, meaning they scale up and down with demand, they are resilient against failures that are inevitable in large distributed systems, they always respond to service requests even if failures limit the ability to deliver services, and they are driven by messages or events from the world around them.

Figure 1 shows an emerging architecture that can meet these requirements.

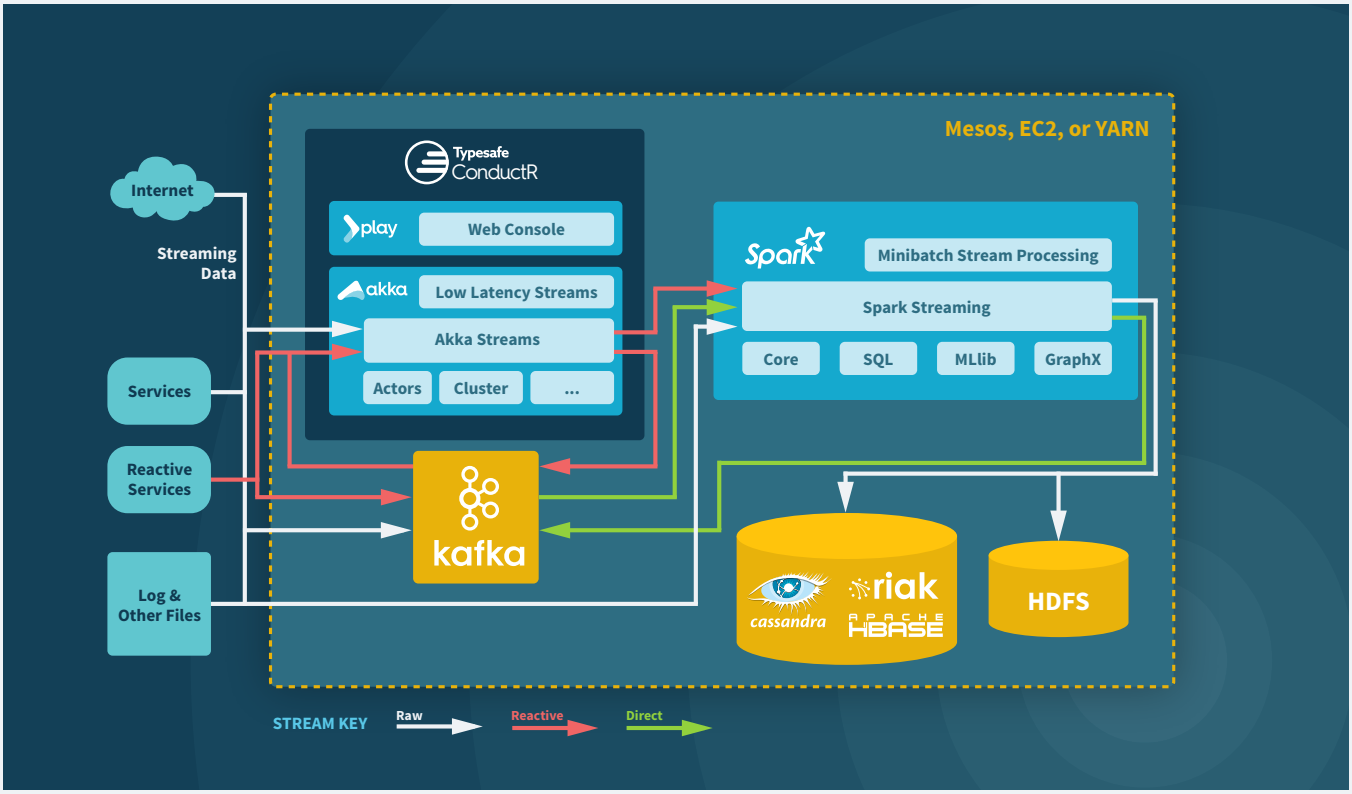


Figure 1

Let's walk through this diagram.

The major components and the triad requirements they satisfy are the following. Not all are shown in the figure:

- **Apache Kafka** for reliable ingestion of high volume streaming data, organized by topics (like a typical message queue). If you are using Amazon Web Services (AWS), **Kinesis** is an alternative. (#1)
- **Cassandra**, **Riak**, **HBase**, or similar scalable datastore with data models and query options appropriate for the application. Sometimes replaced with or complemented by a distributed file system, like **HDFS**, **MapR-FS**, or **Amazon S3**, where simple, scalable, reliable storage of files is required. (#2)
- **Spark Streaming** for flexible ETL (extract, transform, and load) and analytics pipelines, using a *mini-batch* processing model, which can exploit the other Spark libraries, the core (batch) API, SQL/DataFrames, MLlib (machine learning), and GraphX (graph representations and algorithms). (#3)
- **Akka** for real-time event processing, i.e., per-event handling within tight time constraints, with rich integrations to a variety of third-party libraries and systems. Event processing systems specifically designed for Big Data scenarios, but with less flexibility than Akka, include **Apache Samza**, an event processing engine that uses Kafka for messaging and YARN for process management, and **Apache Storm**, a distributed event-processing engine. On AWS, Kinesis provides a library analogous to Samza. (#1 and #3)
- **Typesafe Reactive Platform** (RP), including **Akka**, **Play**, and **Slick**, for implementing web services, user-facing portals, integration with relational databases and other **Reactive** IT services. These distributed systems can be managed with **ConductR**.
- Cluster management infrastructure like **Mesos**, Amazon EC2, or **Hadoop/YARN**, possibly combined with other infrastructure tools, like **Docker**.

When Mini-batch Processing is Sufficient

Hence, there are lots of options. However, for applications where real-time, per-event processing is not needed, where mini-batch streaming is all that's required, **our own research** shows that the following core combination is emerging as the most popular: **Spark Streaming**, **Kafka**, and **Cassandra**. We found that 65% of survey respondents use or plan to use Spark Streaming, 40% use Kafka, and over 20% use Cassandra.

Spark Streaming ingests data from Kafka, the databases, and sometimes directly from incoming streams and file systems. The data is captured in mini-batches, which have fixed time intervals on the order of seconds to minutes. At the end of each interval, the data is processed with Spark's full suite of

APIs, from simple ETL to sophisticated queries, even machine learning algorithms. These other APIs are essentially batch APIs, but the mini-batch model allows them to be used in a streaming context.

This architecture makes Spark a great tool for implementing the ***Lambda Architecture***, where separate batch and streaming pipelines are used. The batch pipeline processes historical data while the streaming pipeline processes incoming events. The result sets are integrated in a view that provides an up to date picture of the data. A common problem with this architecture is that **domain logic is implemented twice**, once in the tools used for each pipeline. Code written with Spark for the batch pipeline can be used for in the streaming pipeline, using Spark Streaming, thereby eliminating the duplication.

Kafka provides very scalable and reliable ingestion of streaming data organized into user defined topics. By focusing on a relatively narrow range of capabilities, it does what it does very well. Hence, it makes a great buffer between downstream tools like Spark and upstream sources of data, especially those sources that can't be queried again in the event that data is lost downstream, for some reason.

Finally, records can be written to a scalable, resilient database, like **Cassandra**, **Riak**, and **HBase**, or to a distributed filesystem, like **HDFS** and **S3**. Kafka might also be used as a temporary store of processed data, depending on the downstream access requirements.

When Real-time Event Processing is Required

If real-time event processing is required, additional components must be added.

Akka comes into play when per-event processing and strict response times are important, either using Akka's **Actor Model** or the new **Akka Streams**, an implementation of the **Reactive Streams** standard, which we'll discuss in more detail later. For example, Akka might be used to trigger corrective actions on certain alarm events in a data stream, while Spark Streaming is used for more general analysis over the stream. The rest of the **Typesafe Reactive Platform** comes into play when building and integrating other IT applications, such as web services, with fast data systems. For example, clickstream traffic processed by an ecommerce store implemented with Play might be streamed to Spark for trend analysis. **ConductR** is an operations tool for managing reactive applications.

Alternatively, **Apache Storm** remains popular, while **Apache Samza** is relatively new for per-event processing. However, most streaming applications don't require stringent, per-event processing, and the attraction of having one tool for both batch and stream processing is compelling. Hence, we expect many applications to migrate to Spark Streaming.

The Flow of Data

The colored lines indicate the kinds of streaming data connections. *Raw* streams (white lines) include sockets (TCP or UDP, including HTTP) and file input (e.g., reading data files dropped into a staging

directory). They contrast with *reactive streams* (red lines), a protocol for supporting dynamic *back pressure*, a mechanism for greater resiliency. Finally, there is a special "direct" connection API (green lines) between Kafka and Spark Streaming. Not all possible pathways are shown, to minimize congestion. Instead, the figure focuses on the most important for Fast Data at scale.

On the left of Figure 1 we show several kinds of data sources, including external sources like the Internet, other services internal to the environment, and data files, like log files.

This data is ingested by Akka, with reactive streams connecting directly to Akka streams, if per-event processing is required, or an alternative like Storm. Otherwise, the data will be ingested into Kafka or sometimes directly into Spark Streaming.

Using Kafka as a durable buffer at the entry point has several advantages. First, it smooths out spikes in the input stream, which can be problematic if fed directly into a more complex pipeline like an Akka or Spark Streaming app. Second, should a downstream processing app crash or suffer a partial data loss for some reason, the data can be reread from Kafka. Hence, Kafka can be used to implement at least once or at most once delivery semantics. *Exactly once* semantics can be implemented using *at least once* semantics in many circumstances with appropriate application logic. For example, unique or incrementing identifiers in the data can be used to filter out repeat messages. A more robust strategy is to use update operations that are *idempotent*, where repeated application of a given update has no additional effect on the state.

Kafka can also be used as a multiplexer-demultiplexer, as illustrated schematically in Figure 2.

Figure 2

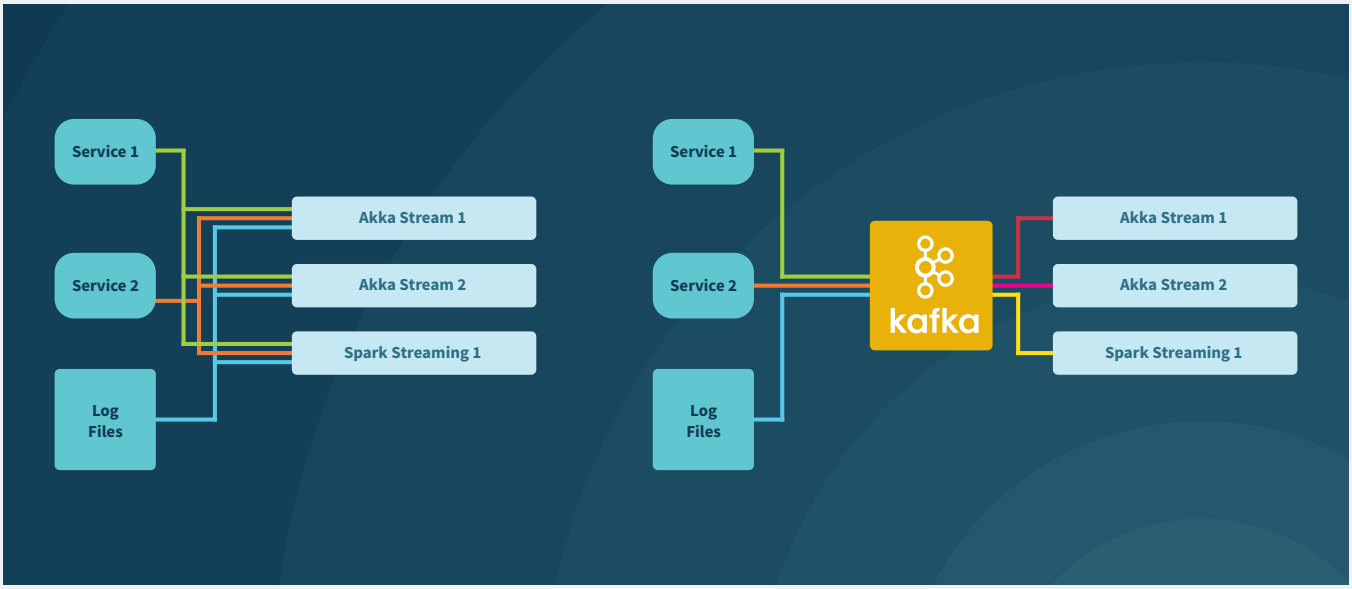


Figure 2: Kafka as a Multiplexer-Demultiplexer

Rather than having ***NxM*** direct connections between components, which are hard to manage, funnel traffic into Kafka, using topics to aggregate data of shared interest. Hence, the overall pathway architecture is simplified.

Finally, all these services can be organized into statically-managed configurations on "bare metal", but that has drawbacks. First, static configurations cannot scale elastically with load, meaning either resources are underutilized during quieter periods or overwhelmed during busy periods. Also, any node or service crash undermines availability, unless some sort of failover is arranged. Hence, it's becoming popular to use a cluster resource manager. **Amazon Web Services**, **Google Cloud Platform**, and **Microsoft Azure** are popular cloud platforms, of course, supporting dynamic scaling of virtualized resources. **Mesos** provides on-premise or cloud-based cluster management, integrated with all the Fast Data components we've discussed. It has the virtue of not interjecting a virtualization layer between the application and the hardware. Hence, performance of the application, once running, is more predictable. **Hadoop** is a mature data platform with integrations to most of these components, as well, but **Hadoop YARN** can only manage Spark jobs, not the other services.

Let's now explore some of these topics in more depth.

Streams

We mentioned the **Reactive Streams** protocol standard for dynamic back pressure. Let's understand what this means and why it's important for Fast Data.

What Are Streams?

We should define the notion of a stream first. For our purposes, a stream is a continuous source of data with an unknown and possibly unbounded size. As long as data continues to be available, we can process it in "chunks" at a time. Stream processing usually imposes requirements on the latency of processing and storage requirements, because of real-world pragmatic issues and application needs.

Conventional Streams

Most stream processing systems will be long-running, for days, months, even years. These systems will see wide fluctuations in the rate of data they ingest from occasional spikes to secular increases and decreases in flow rates. Anything rare becomes commonplace if you wait long enough. Hence, streaming systems are at greater risk of failure due to unusual traffic patterns. You could attempt to build a stream with enough capacity to handle the most extreme spikes without failure, but most of the time the excess

capacity would go unused and be wasted.

Inserting a buffer in front of a stream smooths out those spikes. The queue grows when the producer is currently outputting more data than the consumer can handle and shrinks when the consumer is catching up. The problem is, if this queue is unbounded and the stream runs long enough, inevitably a long-enough period of large traffic will cause the buffer to grow until it exhausts available memory, resulting in a crash.

Memory exhaustion can be avoided by making the buffer bounded, but that does nothing to prevent the initial problem of a long imbalance between the production and consumption rates. Once the buffer is filled, the stream will have to make an ad hoc decision about what to do. Should it drop new data? Should it drop old data? Should it start sampling at a rate it can service? Chances are the stream can't make the right decision because it doesn't have knowledge of the larger context in which it's operating. What's needed instead is system-wide congestion management.

Reactive Streams

Instead, we need a mechanism that uses bounded buffers, but prevents them from filling. That mechanism is *dynamic backpressure*. In the **Reactive Streams** standard, an out-of-band signaling mechanism is used by the consumer to coordinate with the producer. When the consumer can service the load, a push model is used, where the data is simply pushed by the producer as soon as it's available. When the consumer can't keep up, a *pull* model is used, where the consumer requests new data.

Figure 3 illustrates how a reactive stream works.

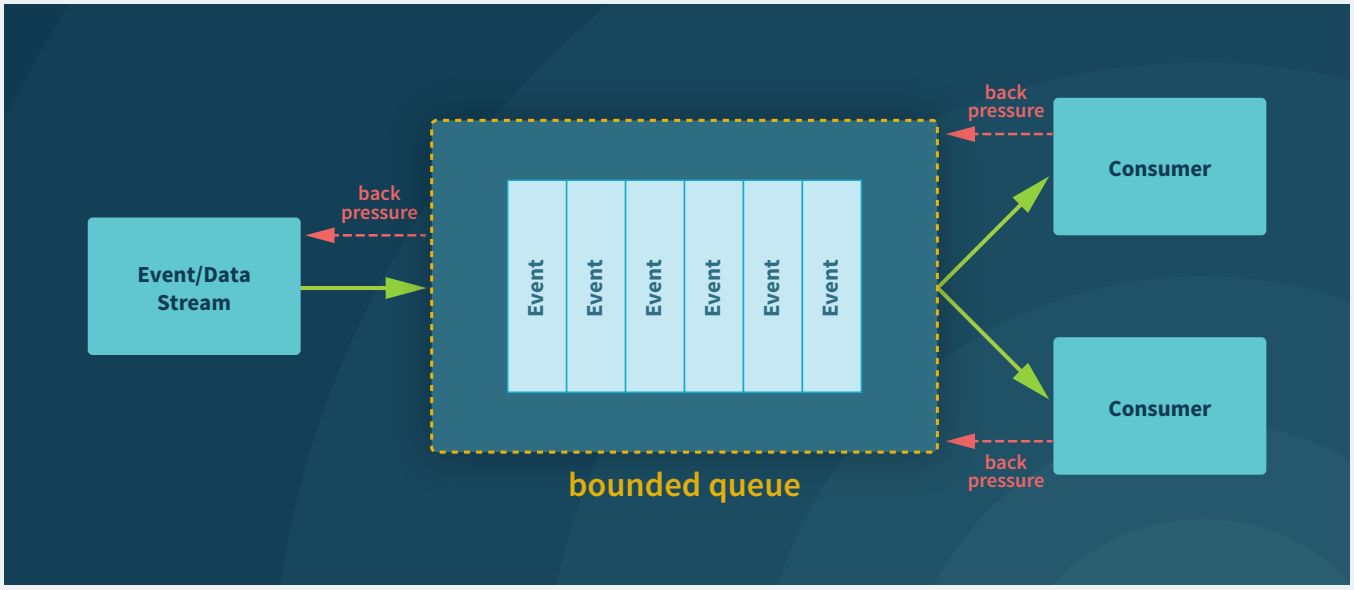


Figure 3

Figure 3: Reactive Steam

This coordination *composes*. If one reactive stream feeds a second one, and the latter switches to a pull model to control the congestion, the first reactive stream may also need to do this, so it doesn't become overwhelmed.

Hence, a graph of reactive streams prevents any one stream from either crashing or making arbitrary choices about congestion management. This structure gives the system designer the ability to make strategic decisions about congestion management at the edges of the graph, while inside the graph, the streams behave reliably. At the margins, data might be dropped or sampled, it might be flushed to temporary storage to be replayed later, or additional processing streams might be started to share the load.

Implementations of the Reactive Stream standard can be found in **Akka**, **RxJava**, and other APIs.

Reactive Streams in Spark Streaming?

Long-running Spark Streaming jobs are already used in many production environments. Spark Streaming has made a lot of progress in recent releases to improve robustness and minimize data loss scenarios. However, support for congestion management is still limited.

Typesafe is contributing improved backpressure support to Spark Streaming:

- Flexible Rate Control & Back Pressure, inspired by Reactive Streams (planned for Spark v1.5).
- An implementation of the Reactive Streams Consumer standard (but not the producer side of the standard), so any reactive stream producer can connect directly to Spark Streaming, including those implemented with Akka Streams, RxJava, and other libraries (v1.6?).

The second piece would be an add-on component analogous to the currently-available support for Kafka, the Twitter "firehose", ActiveMQ, etc. Once available, Spark Streaming can participate in a graph of reactive streams for system wide congestion management.

Reactive Systems

Long-running services, like stream processors, must be reliable. The **Reactive Manifesto** defines a set of four, interrelated traits that are characteristic of reliable, distributed systems, while leaving wide flexibility in how they are implemented. These traits inspired the subsequent work on reactive streams. Figure 4 shows these traits.

Figure 4

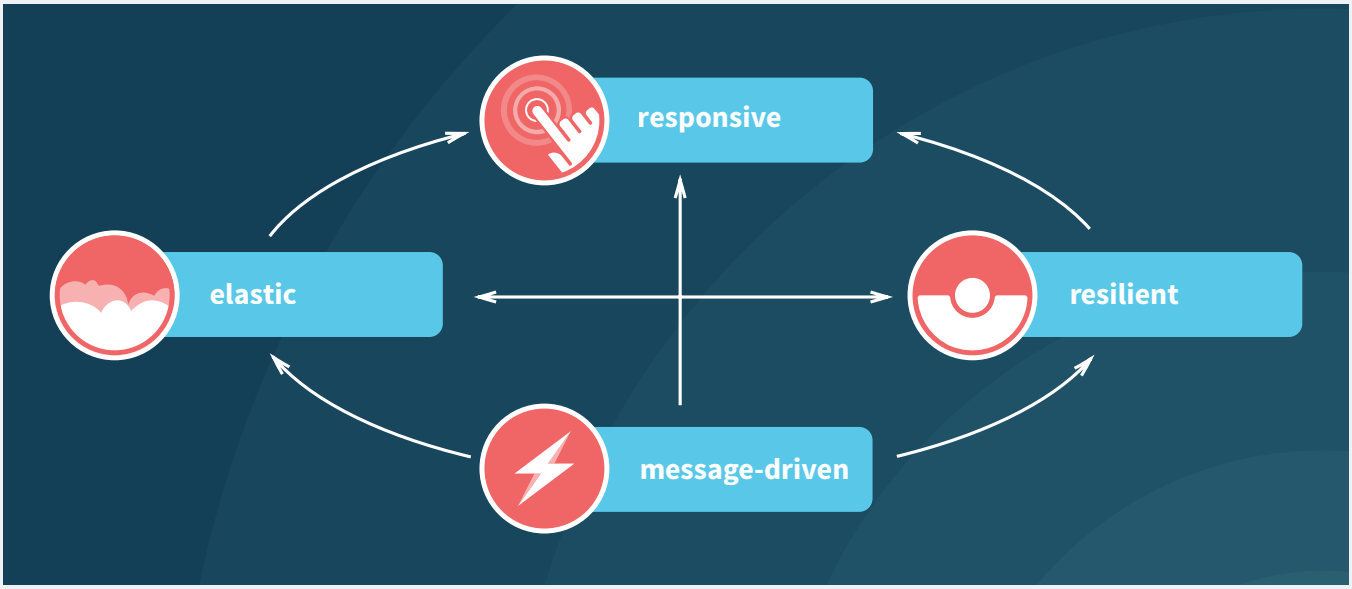


Figure 4: The Four Reactive Traits

- **Responsive**: A request for service is always answered, even when failures occur.
- **Elastic**: Services scale up and down on demand.
- **Resilient**: Services recover from failures.
- **Message Driven**: Services respond to the world, not attempt to control what it does.

Long running services in a dynamic world must be able to scale up, as well as down, on demand. Scaling down is often harder to implement. Here, *up* and *down* are measures of size. The actual scaling is *horizontal*, across multiple nodes, rather than *vertically*, within a single machine.

Node or service outages, network *partitions*, and other problems are certain to happen in any nontrivial distributed system that runs long enough, yet the system needs to remain responsive to client requests for service, if when all it can do is respond, “I can’t help you now”. This requires that the notion of

networks must be a “first class”, expected concept in the system, including all the uncertainties that are inherent in networking.

Ideally, systems recover from failures quickly. A logical service may failover to backup instances. Failed service instances may be restarted automatically. Hence, the notion of failure must also be a first class, expected concept in the system. They are normal, anticipated, and handled.

Finally, to truly react to the world around it, the system can’t be “command and control”. Rather, it must respond to stimuli from the world around it. This means it must be driven by messages. Why is the word “message” used and not “event”? An event is something that happens, while a message has a specific sender and receiver. A message may carry an event as payload, but the notion of a message is more appropriate as a system architecture concept, where communication between subsystems is the goal.

The systems in Figure 1 implement some or all of these traits in various ways. Weaknesses in some of the systems can be complemented with others. For example, Spark Streaming’s mini-batch model is not designed to be a message-driven system, but Akka and Kafka provide this capability.

Going Deeper into Spark and Spark Streaming

Spark started as a batch-mode computation engine, much like the predecessors that inspired it, **MapReduce** and **Microsoft’s DryadLINQ**. However, stream processing has grown in importance in the last few years. Could Spark be adapted to support it?

The Mini-batch Model of Spark Streaming

Because Spark can execute tasks efficiently, the Spark team realized that a straightforward extension to support streaming is to capture data in time slices and process each one as a “mini-batch”. Hence, with relatively little effort, Spark was extended to support streaming with the side benefit of making the other Spark APIs reusable in a streaming context, including the core **RDD API** (RDD - Resilient Distributed Dataset, the core data structure that partitions data across the cluster), **SparkSQL** with the new **DataFrame** abstraction, **MLlib** for Machine Learning, and **GraphX** for graph representations and algorithms.

The tradeoff for the mini-batch model is higher latency from the time data arrives to the time that information is extracted from it, a few seconds to minutes, vs. sub-second responses possible with per-event systems, like Akka and Storm. The mini-batch model means that Spark Streaming is not designed for individual event handling either.

Still, the majority of applications for streaming don’t require per-event, low-latency processing. Most

applications just need to reduce the hours of latency for typical batch jobs to seconds or minutes, for which Spark Streaming is an ideal fit.

Batch Reborn: The Triumph of Functional Programming and Scala

Batch processing remains important and Spark's rapid rise in popularity has been driven primarily by its advantages over MapReduce. Spark's success is not just the result of clever engineering decisions by smart people. It is due in part to the ideas of *Functional Programming*, as exemplified in Scala, which are ideal for working with data, big or otherwise.

Big Data and Functional Programming

Functional Programming (FP) is actually an old body of ideas, but it was primarily of interest in academic circles until about 10 years ago. Until that time, scaling applications vertically, i.e., with ever faster hardware, had been a tried and true technique, but the need to scale horizontally grew in importance for scalability as Moore's Law reached a plateau, and to achieve greater resiliency when individual services or hardware fail.

FP emphasizes several traits that are inspired by Mathematics, which are ideal for writing robust, concurrent software:

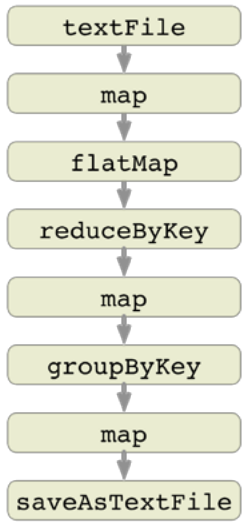
- **Data should be immutable:** Mutating data is convenient, but we now know it is the most common source of bugs, especially in concurrent applications.
- **Functions should not perform side effects:** It's easier to understand, test, and reuse functions that don't change state outside themselves, but only work with the data they are given and return all their work to the caller.
- **Functions are first class:** This means that functions can be used like data, i.e., passed to other functions as arguments, returned from functions, and declared as values. This leads to highly composable and reusable code, as we'll see in a moment.

While FP proved fruitful in making concurrent applications easier to write, the growth of Big Data over the last ten to fifteen years has accelerated interest in FP, because those same Math-inspired processes are the ideal way to think about data.

Actually, this fact is also not really new, as the venerable **Relational Model** and **SQL** databases are based on a branch of Mathematics called **Set Theory**, producing one of our most successful and long-lived technologies we have. In this sense, SQL is a subset of the more general capabilities of FP for data

manipulation. Loosely speaking, data problems can be considered either query problems, for which languages like SQL are ideally suited, or as **dataflow programming**, where data is passed through a graph of processing steps, each of which provides transformation, filtering, or grouping logic. Combined together, the graph transforms an input dataset into an output result.

Consider Figure 5, which illustrates a dataflow for the Inverted Index algorithm, along with Spark API code for implementing it:




```
sparkContext.textFile("/path/to/input")
.map { line =>
  val array = line.split(",", 2)
  (array(0), array(1))
}.flatMap {
  case (id, contents) => toWords(contents).map(w => ((w,id),1))
}.reduceByKey {
  (count1, count2) => count1 + count2
}.map {
  case ((word, path), n) => (word, (path, n))}
.groupByKey
.map {
  case (word, list) => (word, sortByCount(list))
}.saveAsTextFile("/path/to/output")
```

Figure 5

Figure 5: Inverted Index Dataflow

Simply stated, *Inverted Index* ingests a data set of document identifiers and their contents, then tokenizes the contents into words and outputs each word as a key and a list of the document identifiers where the word is found. Usually, the count of occurrences of the word in the document is paired with the identifier. The index is *inverted* in the sense that the output words are keys, but in the input, they were part of the values associated with document identifier keys.

An example is a basic web search engine, where the input data set is generated by web crawlers. The document identifiers might be URLs and the contents might be the HTML for each page found.

To illustrate the results, here are some fictitious records that might result from running Inverted Index on web crawl data:

Word	DocIDs and Counts
Akka	**(akka.io, 1000), ..., (typesafe.com, 400), ...**
Spark	**(spark.apache.org, 2000), ..., (databricks, 500), ..., (typesafe.com, 250), ...**
SQL	**(oracle.com, 5000), ..., (mysql.org, 4000), ...**
...	**...**

Using the Spark API, Figure 5 shows one possible implementation of the algorithm. We won't discuss all the details. What's important is the overall picture of modeling a dataflow as a sequence of steps that translate directly into a concise program.

However, for completeness, here is a brief description of this code. First, we assume that the input records are actually comma-delimited text, one line per document identifier and its contents. The **`textFile`** method is used to read the lines, then map is used to split each line on the first comma, yielding the **`id`** and the contents. Next the contents is tokenized into words (the implementation of **`toWords`** is not shown), and the (**`w, id`**) combination (a "tuple", where **`w`** is short for "word") is used as an intermediate key with a value of 1, the "seed count" for counting the unique (**`word, id`**) combinations. The **`flatMap`** method is an extension of **`map`**, where each iteration yields a sequence of zero to many output tuples, which are flattened into one long sequence of tuples from all the lines. The counting is done by reduceByKey, which is an optimized version of a SQL-like "group by" operation, followed a transformation that sums the values in the group, the seed counts of 1.

At this point, we have records with unique (word, id) keys and counts >= 1. Recall that we want the words alone as keys, so the last steps perform the final transformations. The next map call shifts the tuple s tructure so that only the word is in the first position, the "key position" for the groupByKey that follows, which is a conventional "group by" operation, in this case joining together the records with the same word value. The last map step takes each record of the form **`(word, list((id1, n1), (id2, n2), (id3, n3), ...))`** and sorts the "list" of **`(id, n)`** pairs by the counts, descending (**`sortByCount`** function not shown). Finally, we save the results as one or more text files.

Once you understand how to divide a algorithm into steps like this, you want to translate that design into code as concisely and effectively as possible. The Spark API achieves this goal beautifully, which is why we are enthusiastic supporters of Spark. No doubt many other adopters feel the same way.

By the way, when we discussed earlier how *first* class functions are important, that's a crucial feature we exploited in this code example. All those "{...}" blocks of code are anonymous functions that we passed as arguments to **`map`**, **`flatMap`**, **`reduceByKey`**, etc. to compose the behaviors we needed.

Why Spark Was Built with Scala

The Spark Core API is patterned on the **Scala Collections API**, which is itself an object-oriented realization of classic collections APIs, like those found in purely functional languages, e.g., **Haskell** and **OCaml**. Martin Odersky, the creator of Scala, **recently called** Spark, "The Ultimate Scala Collections."

In 2008–2009 when Spark was started as a research project at the University of California at Berkeley by Matei Zaharia, he recognized that Scala was an excellent fit for his needs. Here's an answer he once gave for "Why Scala?"

> **"Quite a few people ask this question and the answer is pretty simple. When we started Spark, we had two goals—we wanted to work with the Hadoop ecosystem, which is JVM-based, and we wanted a concise programming interface similar to Microsoft's DryadLINQ (the first language-integrated big data framework I know of) on the JVM, the only language that would offer that kind of API was Scala, due to its ability to capture functions and ship them across the network. Scala's static typing also made it much easier to control performance compared to, say, Jython or Groovy."**
>
> **Matei Zaharia**
> Creator of Spark, CTO and Co-founder, Databricks

The subsequent growth and runaway success of Spark has validated his choice of Scala, which remains the primary implementation language for Spark as well as one of the five languages supported in the user-facing API (Scala, Java, Python, R, and SQL).

Scala

To recap the benefits of Scala for Fast Data, it makes it easy to write concise code and it provides idioms that improve developer productivity. Scala is a JVM language, so applications can exploit the performance of the JVM and the wealth of third-party libraries available.

Finally Scala is a fusion of Functional and Object-Oriented Programming, offering the best of both worlds.

> **"We think that the biggest gains are in the combination of the OO model and the functional model. That's what we firmly believe is the future."**

Martin Odersky
Creator of Scala, Chairman and Co-founder, Typesafe

Martin Odersky created Scala to apply lessons that had been learned from Java's successes as well as its drawbacks, plus exploit the latest results from academic research that promote better quality, reliability, and are better suited for the kinds of problems that developers face today, such as the need to write robust, concurrent software.

Martin firmly believed that a fusion of FP, for it's "correctness" properties, and OOP, for its modularity and encapsulation, was essential for a modern, practical, effective programming language.

Functional Programming is the Killer Paradigm for Data Apps

In a way, Scala was in the right place at the right time, with the right tools to offer. The core functional operations, like the subset we showed in the previous code example, are the result of years of research in functional programming. By exploiting this knowledge, combined with other pragmatic design choices, Scala has been ideal for Spark, and Spark is now driving adoption of Scala. In more general terms, the growing importance of data-centric applications has driven interest in functional programming, leading us to the following conclusion:

Functional Programming is the Killer Paradigm for Data Apps

EXPERT TRAINING

Delivered On-site For Spark, Scala, Akka And Play

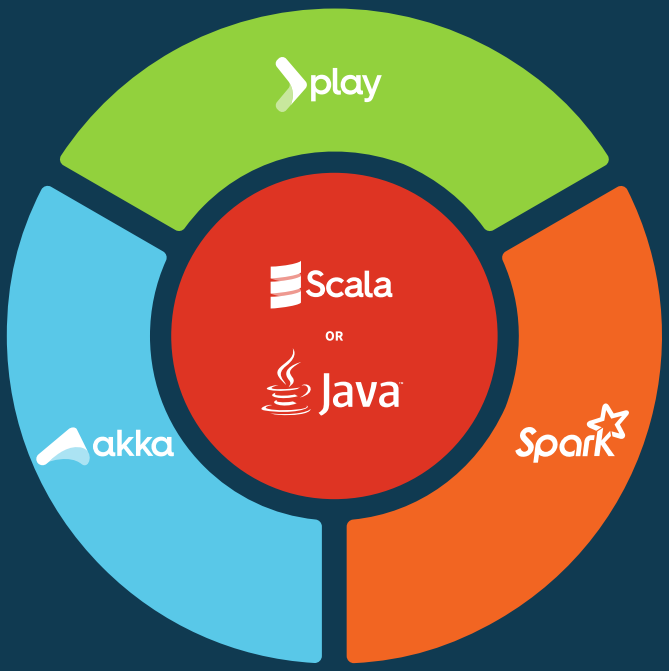
Help is just a click away. Get in touch with Typesafe about our training courses:

- Intro Workshop to Apache Spark
- Fast Track & Advanced Scala
- Fast Track to Akka with Java or Scala
- Fast Track to Play with Java or Scala
- Advanced Akka with Java or Scala

Ask us about local trainings available by 24 Typesafe partners in 14 countries around the world.

CONTACT SALES

Learn more about On-site training

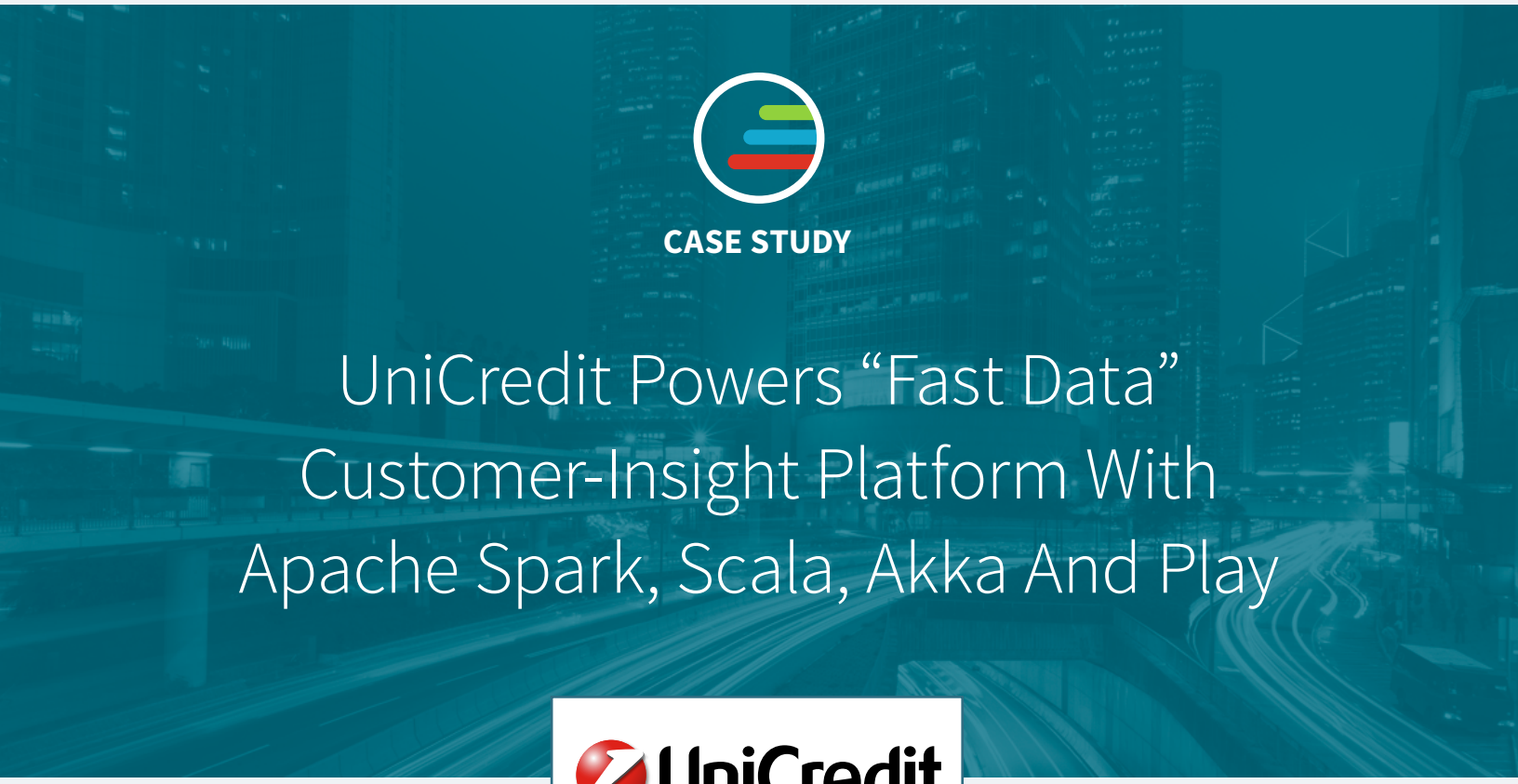


Typesafe

Typesafe (Twitter: @Typesafe) is dedicated to helping developers build Reactive applications on the JVM. With the Typesafe Reactive Platform, developers can create message-driven applications that scale on multicore and cloud computing architectures using Play Framework, Akka, Scala, Java, and Apache Spark. Typesafe contributes to Apache Spark and Spark Streaming and partners with technology pioneers such as Databricks, IBM, and Mesosphere. Typesafe is venture backed by Shasta Ventures, Bain Capital Ventures, Juniper Networks and Greylock Partners with headquarters in San Francisco and offices in Atlanta and Switzerland.

© 2015 Typesafe

Appendix



CASE STUDY

UniCredit Powers "Fast Data" Customer-Insight Platform With Apache Spark, Scala, Akka And Play

UniCredit

Executive Summary

UniCredit is a leading European financial group with an international network spanning 50 markets. With circa 8,500 branches and 147,000 employees serving more than 32 million clients, the Group has commercial banking operations in 17 countries and assets of €900 billion. As one of the strongest banks in Europe, UniCredit has a Common Equity Tier 1 Capital ratio of 10.35 percent pro-forma (fully-loaded Basel III, including Pioneer). It also has the largest presence of banks in Central and Eastern Europe, with nearly 3,500 branches and assets of €165 billion.

Looking forward, UniCredit decided to proactively confront an impending challenge in 2012: how to continue to serve an aging customer base with their existing Java stack, while simultaneously collect valuable insight from their enormous amounts of historical data in order to continually evolve their modern web and mobile platform bank services?

In 2012, an executive decision was made to create a new team called the Group Research and Open Innovation department, tasked with researching innovations that would power UniCredit for the future. After reviewing all the requirements, the Group Research and Open Innovation team selected **Apache Spark** and **Typesafe Reactive Platform** technologies Akka, Play and Scala in order to create a distributed, resilient, fast data processing platform. Within two weeks, a prototype application was ready to test.

The reason to modernize

Facing an inability to easily access and rapidly analyze decades of historical data, UniCredit's team started on their "fast data" project. The goal was clear: unlock the value in this massive quantity of data that they had never before been able to see in order to understand the needs of future customers.

One challenge standing in their way of getting a meaningful, expansive view of their business customers was a mix of legacy data repositories and storage. Some of UniCredit's technologies in use are IBM DB2, Oracle DB, Teradata, Oracle Exadata, and even magnetic tape (as required by the Italian government). The challenge was to find a way to connect everything together meaningfully.

UniCredit was tasked with uncovering and graphing relationships between companies that are clients, looking for patterns or connections that would help them better provide services for interconnected customers.

Going Reactive with "fast data" to understand customer relationships and interactions

The UniCredit team started off by implementing Cloudera's Hadoop distribution and HBase, namely as a way to bring these enormous quantities of disparate data into one place. But when it came down to bringing this data into motion and make it valuable through algorithmic, graphical data analysis, this solution was insufficient. What was needed was a highly-performant data pipeline that would be resource efficient and resilient, and also fun to work with. The main characteristics were:

- **Highly Performant**—Before the introduction and headlines around Spark, the team had started using Hadoop MapReduce and Scalding, a Scala library for specifying Hadoop jobs built by Twitter. When Spark was introduced, the team quickly moved to integrate it with Hadoop to get much greater performance and work with the data more easily. In addition to the new functionality needed for their fast data system, UniCredit now uses Spark for their existing Hadoop jobs, such as crunching data from various legacy systems and graphing page rankings.
- **Mission-critical Level Resilience**—Even though this system is not consumer-facing, it is nonetheless considered mission critical and resilience is a huge factor. For UniCredit's Corporate

Relationship Management department, this system is a veritable "Swiss Army knife" that consolidates dozens of different tools previously used, and without it employees cannot harness the value that this data reveals.

- **Capable of distributed computing**—Use of Akka Clusters proved to be an easy and effective way to deal with distributed data computation for complex processing pipelines, where a "microservice-style" approach to computation is efficiently crunched by clusters of Akka actors.
- **Fun to work with, fast to prototype**—An important factor that is difficult to calculate is developer happiness. Scala is an incredibly concise language that reduces boilerplate, and Play Framework's console and instant code update features made the prototyping process efficient and lean. UniCredit was able to create a prototype in a matter of weeks to share with management.
- **Capable of growing to handle streaming**—With enormous amounts of data to be processed, the potential is there to eventually incorporate data streaming technologies with built-in back pressure like Akka Streams and, soon, Spark Streaming.

With these requirements in mind, the team began looking into new technologies for building a prototype of this system. With some existing experience in Scala, Apache Spark (written in Scala) was chosen as the data computation component, and after seeing the performance and features of Apache Spark working with Akka, the decision was made to go for Typesafe Reactive Platform.

The results: a Fast Data platform that delivers new insights into client relationships

UniCredit's platform is based on distributed Akka clusters to maintain their application's resilience and elasticity. Written in Scala, their "Fast Data" project heavily utilizes Akka, including Akka Persistence and Akka HTTP (formerly Spray) to support distribution and for collecting/sending data from difference sources. Play Framework is used for quick prototyping and RESTful API/HTTP communication. For added performance, the application places Spark alongside Cloudera's Hadoop distribution, in addition to HBase, CouchDB, and Aerospike.

After presenting the prototype, it was tested in production for a few weeks before being declared production-ready and launched into UniCredit's infrastructure. Soon, the insights from this project became so valuable that UniCredit decided to build a new "intelligent CRM" that other departments could integrate and utilize for large-scale analysis.

- **Revealing new insights never before seen**—by selecting a new "Reactive Stack", based on Spark, Akka, Play and Scala, UniCredit was able to access and analyze data sets that previously were

never connected, allowing them to utilize decades of information and develop new services for interconnected corporate clients.

- **Seeing the value**—UniCredit was able to uncover relationships between their corporate customers in the first several weeks, enabling them to understand and generate more personalized services that weren't possible before.
- **Pay-it-forward**—with the proven success of this project, UniCredit plans to use these technologies in more systems across their enterprise.
- **Handles growing needs**—with these technologies, future additions of streaming technologies like Akka Streams and Spark Streaming are not only possible, but simple.

Convinced of the power of Spark, Scala and Akka, UniCredit has another prototype in the works, utilizing even more of the so-called "Reactive Stack" technologies by combining Scala, Akka and Spark with Apache Kafka. In fact, a new experiment using Akka Streams and Spark Streaming for Natural Language Processing (NLP) has begun in order to analyze different types of content on the web.

CASE STUDY

UK's Online Gaming Leader Bets Big On Reactive To Drive Real-Time Personalization

Real-Time Customer Data Becoming New Battleground Within Gambling Industry

For some organizations, "Going Reactive" is a consideration for the future. For industry leaders, Reactive systems have already enabled major competitive advantage.

The United Kingdom legalized online gambling 10 years ago. Although the market for online "punting" is still very young, it's growing at a sharp clip. In the first five years the annual market reached £1.48bn, accounting firm KPMG estimated. By 2014, the UK Gambling Commission confirmed market growth to £2.44bn.

With hundreds of gaming operators fighting for slices of this action, one of the UK's oldest offline gaming operators -- William Hill -- managed not only to enter the nascent online game, it quickly secured an astounding 16% of market share. By 2014, William Hill reported £399m in online gambling revenue, up 175% from the previous year. Now they're number one in online gaming.

So what role does technology play for the growth strategy of the UK's largest online gaming operator?

Doubling Down on New Personalization Features and Functionality

In the online gambling industry, the basic betting services offered by operators are marginally different - and the odds themselves have become so commoditized that many large operators globally even offer free APIs for live odds. Operators compete fiercely for differentiation with user-facing features and functionality. William Hill attributes a large part of its marketshare to simply delivering a better customer experience than other gambling operators in the UK.

"We can't predict the future, but we know if we are not in control of our data, we'll never be in a position to innovate. Understanding the structure of our data and how we can leverage it to deliver highly personalized content to our users is the next major opportunity for revenue growth."

Patrick Di Loreto
R&D Engineering Lead, William Hill

Di Loreto and William Hill have continued to explore ways to grow marketshare on the basis of what the customer wants. They believe their most interesting opportunities are tied to personalization - assigning logical reasoning to customer behavior, and presenting personalized data on the basis of machine learning and intelligent predictions.

For William Hill, "personalization" takes on an even greater importance with the rise of "In-Play" markets - where users can bet on live games, with new betting options presented in real-time (e.g. "will Sharapova win the next point in a tennis match?") In-Play represents the ultimate challenges for personalization. How to make users aware of new betting options that match sports and teams they are interested in is the challenge. William Hill needed computing infrastructure that can instantly draw correlations between user actions on their site, other sites they visit, betting propositions they look at and act on, what other similar players do under similar circumstances, and lots of other reasoning (deductive reasoning, inductive reasoning and abductive reasoning), all in a blink of an eye.

"Many of our new, instant campaigns need to feature personalized recommendations that may only be valid for thirty seconds, and all that must perform perfectly during special events and namely on Saturday afternoons, when we regularly face up to 100x traffic peaks."

Patrick Di Loreto
R&D Engineering Lead, William Hill

A Winning Hand of "New Stack" Technologies Fuel William Hill's Data Streaming and Path to More Personalization

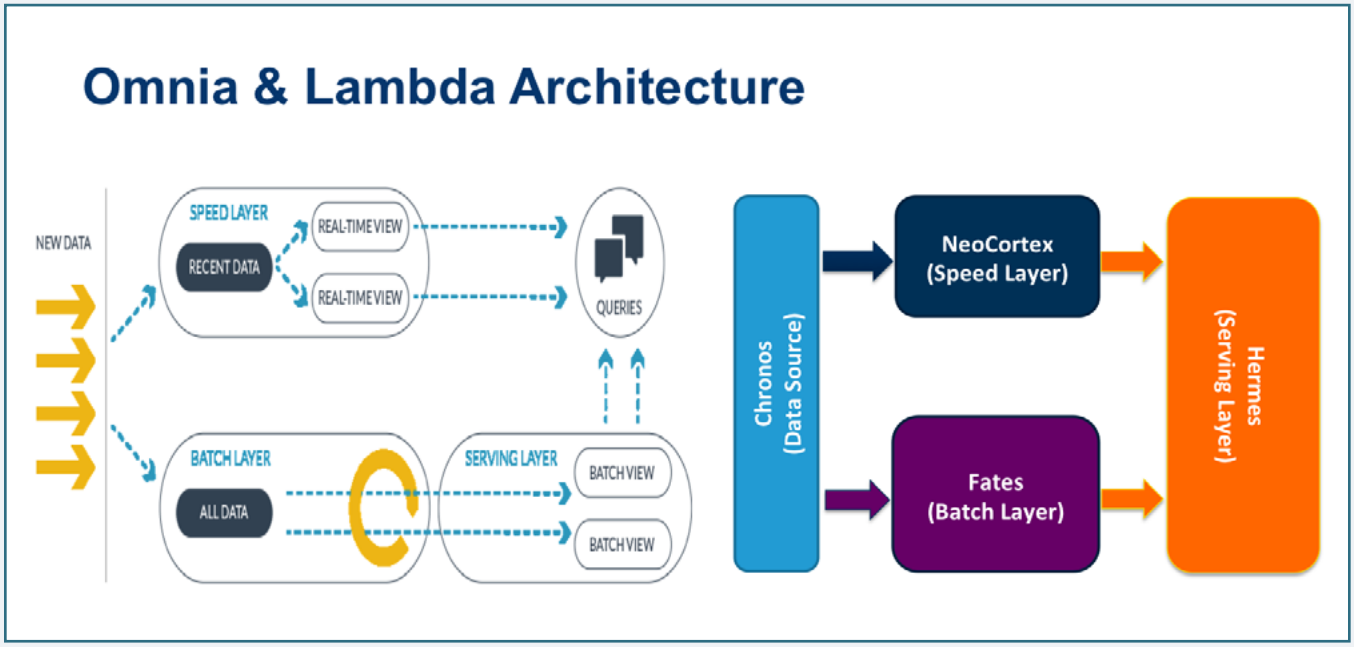
To capture, correlate and understand in real-time every customer interaction leveraging the power of machine learning, Di Loreto realized his team had to re-imagine its application architecture to deliver on the realities of all of the data in motion and logic required to exploit it. As core technologies for data streaming–like Spark, Cassandra, and Kafka–came into view, it became very obvious that the world with Java, Hibernate and Spring was not going to be able to deliver the right foundation for bringing more personalization opportunities to the business.

"Our journey to Reactive started with embracing core functional programming concepts and continues with a more responsive and resilient system. Really we saw that a change in how we thought of system architectures was going to be a prerequisite to 'Going Reactive, and after a crash course in Scala we decided to use it to prototype Omnia. We felt it would let us handle all the data wrangling and logic required to manage the data streams on our users. It was clear that core elements of functional programming—monoids, observable subgroups, location transparency—were must-haves."

Patrick Di Loreto
R&D Engineering Lead, William Hill

As William Hill embraced functional programming, they quickly arrived at the combination of Scala and Akka as the backbone of their application infrastructure. With Scala handling the most difficult issues of concurrency, and Akka Actors doing the heavy lifting around location transparency and negotiating with other key distributed frameworks - William Hill is quickly building a set of tools at the data layer, to create a foundation for long term success in personalization features and functionality.

Getting to Know William Hill's Home-Grown Data Streaming Frameworks Written on Top of Scala, Akka, Spark, Kafka, Cassandra, and Mesos



- **Omnia**—The primary production data platform, collecting user information from all activities on the website, as well as online and offline events related to the customer.
- **NeoCortex**—a home-built Spark framework that allows developers to write applications that are simple to read, and spares them from the complexity of distributed computing and multithreading.
- **Chronos**—the Play Framework-based application that collects user activity both inside and outside the website and connects to all applications internally and externally (including Facebook and Twitter).
- **Fates**—another custom-built framework that correlates specific activity streams from Chronos and makes them configurable as timelines of customer events.
- **Hermes**—the service layer that makes all the data managed by Omnia available to B2B and B2C applications.

A "New Stack" for a Customer Data-Driven Business

As William Hill builds the new "Reactive Stack" for its customer data-driven business, it has been an aggressive adopter of complimentary distributed frameworks that play very nicely with the key Reactive tenets of Scala and Akka, and make the company's data platform Reactive all the way through. Each technology assumes a different role that compliments the others.

Scala

- Functional by design, brings core characteristics for concurrency, and doing things in a declarative fashion.
- Brings concurrency to all systems that William Hill builds (where Java concurrency was overly complex, Scala has programming abstractions that make it easier to write concurrent code).
- Di Loreto calls Scala a "prerequisite" to going Reactive; from his point of view, the observable and Reactive elements to William Hill's' system start with monoids, and Scala is the key.

Akka

- Along with Scala, serves as the backbone of Omnia and other core data frameworks at William Hill; allows the team to program the software and not care about where the components live physically.
- Handles key aspects to clustering and sharding functionality that allows information in the Omnia platform to be equally distributed inside of a cluster.
- Heavy use of Akka Persistence and Actors connecting other frameworks brings resilience, fault-tolerance and removes any single points of failure to William Hill's data platform.

Spark

- The data computation engine that handles all of the analytics of user information.
- Works very fluidly with Cassandra (the primary datastore in William Hill's data streaming architecture).
- Core to real-time processing, logical reasoning, graphing and other heavy lifting related to correlating different datasets related to William Hill users.

Cassandra

- Distributed database, manages storage in a Reactive manner and creates the timelines that get passed to Spark for logical reasoning (very solid integration with Spark for CQRS systems, with mature connectors).

- Brings high availability and embraces the view that failure is a normal part of a system, to ensure that no single failures can bring the system down.
- Handles “horizontal scale” extremely reliably / efficiently.

Mesos and Docker

- All of William Hills’ new data architecture is written as microservices that are contained in Docker containers and run on top of Mesos.
- Mesos manages tens of Docker containers - launching, managing, provisioning resources.
- Brings elasticity to the cluster, CPUs, memory, I/O and other infrastructure, and adds portability of infrastructure, convenient packaging and deployment of services with isolation from other services.

The Results - Unshakable Resilience at the Speed of Real-Time Big Data

For William Hill, time is truly of the essence. Knowing the latest details for the most popular events–and being prepared to handle massive spikes in traffic during events–is a key differentiator both in their home UK market and worldwide.

Maintaining resilience in the face of 100x peaks in traffic is something that William Hill deals with on a weekly basis. For example, Mondays and Fridays are typically “slow days”. Tuesdays and Thursdays frequently see 20x the traffic of slow days, and Wednesdays (when Champions League is on) can reach 50x the traffic. However, it gets interesting on Saturdays in the UK between 3 - 5pm, when William Hill regularly sees a 100x increase compared to the previous day, and drops off again significantly on Sundays.

Given the millions of customers that William Hill serves, handling this type of load required a highly performant framework that could be distributed across different systems and handle millions of parallel requests. William Hill utilizes Spark on top of Akka clusters for exactly this: Akka serves as a transparent processing framework for all requests, enabling Spark to process the real-time data, using Kafka and Cassandra for messaging and storage.

“Simply, without Omnia using Akka and Spark to handle distribution and speed, none of these services would have been possible to launch due to our real-time requirements and extreme peaks in load.”

Patrick Di Loreto
R&D Engineering Lead, William Hill

A Reactive Foundation for Future Microservices

Patrick and his team launched Omnia on July 21, 2015 in order to prepare for their peak season, which goes from September to May each year. Along with the July launch comes two services that William Hill are counting on to engage their clients like never before:

1. **Recommended "Bet Suggestions"**: Omnia's recommendation engine works in a similar way to Amazon's media suggestions, with the major difference being that compared to a discount on a book, William Hill's "Bet Suggestions" are fleeting and may only last seconds. The need to concurrently handle millions of recommendations with very fast lifetimes would have been impossible without Akka actors and Spark.
2. **Gamification**: William Hill wants to explore whether gamifying the experience outside betting and gambling leads to higher engagement and have designed a feature that rewards customers for performing specific actions in a short amount of time, sometimes lasting only seconds. These activities include personalized challenges specific to the event or sport the customers wants. Not only is it challenging to provide these services, but the ability to concurrently process it all during traffic peaks is also needed. Millions of customers are performing actions and William Hill needs to be able to provide relevant challenges and also reward the customers in real-time during busy hours.

Since the launch, Omnia has been available to VIP clients, so the real test of Omnia comes on August 8th, when William Hill is planning for a pre-season peak during the UK's Premiership League.

To prove that Omnia can handle large audiences, at 3pm on the Premiership League opening day William Hill will test the recommendation engine by sending out 500,000 personalized offers, most with life-spans of less than 5 seconds, to users on their platform.

> **"The most important feature for Omnia right now is to cope with large numbers of simultaneous, real-time offers. This is the first chance for us to see how Omnia reacts to this new type of services, where previously we couldn't even consider without these new technologies powering it all."**

Patrick Di Loreto
R&D Engineering Lead, William Hill

Needless to say, William Hill is investing heavily in the technologies powering their Omnia platform and has instituted an innovation team in the tech startup haven of Shoreditch in London. The goal: explore new concepts and features, mostly by leveraging Omnia's reactive capability, to make brand new products and services that continually embrace Reactive technologies.