**Indian Institute of Information Technology-Allahabad**



PROJECT

SEMESTER – VIII

**Distributed framework for back-testing and**

**Post Trade Analytics**

Submitted By:

Mayank Vijay

IIT2013127

**CANDIDATES’ DECLARATION**

I hereby certify that the work which is being presented in the B.Tech. Project Report entitled “**Distributed framework for back-testing and post trade analytics**”,being submitted asa part of VIIIth Semester Project Evaluation to the Department of Information Technology of Indian Institute of Information Technology, Allahabad, is an authenticated record of my original work in **Edelweiss Financial Services Ltd** from February 2017 to June 2017 carried out under the guidance of Mr. Gaurav Shah.

**Date:** 25th July 2017

**Place**: IIIT-Allahabad

**Student’s Name**: Mayank Vijay

**Table of Content**

**TABLE OF CONTENT**

Abstract ……………………………………………………………………….. 4

1. Introduction and Motivation ………………………………………. 8 - 12
2. Problem Definition…………………………………………………. 12 - 14
3. Literature Survey ………………………………………………… 14 - 16
4. Proposed Methodology ………………………………………… 16 - 26

4.1. Feature Extraction …………………………………………… 16 - 20

4.1.1. SIFT …………………………………………………... 17 - 19

4.1.1.1. Scale Space Extrema Detection …………………... 17

4.1.1.2. Keypoint Localization …………………………….. 18

4.1.1.3. Orientation Assignment ………………………….. 19

4.1.1.4. Keypoint Descriptor …………………………….... 19

4.1.2. Advantages of SIFT ……………………………………... 19

4.2. Vocabulary or Codebook Generation ……………………………... 20

4.2.1. KMeans Clustering Algorithm …………………………….. 20

4.3. Training Classifier …………………………………………….. 21 - 22

4.3.1. SVM ………………………………………………………. 22

4.4. Pseudocodes ………………………………………………… 23 - 25

4.5. Algorithms used for comparison ……………………………… 25 - 26

4.5.1. HOG ……………………………………………………… 26

**ABSTRACT**

At Global Markets, we have several business requirements that require very quick analytics on huge data that helps the researchers and traders in making big profits.

Big data is really promising and differentiating for financial services companies that rely heavily on making money from money than selling products. With no physical products to manufacture, data is one of arguably their most important assets. The business of banking and financial management sector, especially if you work for department that thrives on profits in stock marketsis where transactions are conducted in millions everyday, is driven by the stock market data received from the National Stock Exchange everyday during the market hours.

Simultaneously researchers apply different algorithms on it to make more and more profits for the company and traders trade heavily which also generates huge data everyday. Thus, the dataset is really huge which needs to be handled really efficiently as the business relies on back testing and analytics, for the creation of strategies and testing these strategies for authenticity and profit making ability. The performance and the ability to judge the optimal strategy greatly increase, as the dataset utilized in the build and back test strategies increase. Thus, handling Big Data is of at most importance for these firms.

The methods of data collection and analytics were traditionally used through excel and sheets for the ease of use fail to compete with the sheer size and variety of this data, thus it is required to design a system that can handle these humongous amounts of data. The best and optimal solution to tackle this problem is the creation of distributed systems and setting up frameworks that can work on distributed environment which are fast, fault-tolerant and resilient. This, project aims to build such a system of distributed network, where in humungous amounts of data can be stored, processed and analysed.

**INTRODUCTION**

At Global Markets we deal with various kinds of Algorithmic and Systematic Trading. Systematic trading is a way of defining trade goals, risk controls and rules that can make investment and trading decisions in a methodical way. Algorithmic trading is pre-programmed trading instructions to buy/sell shares to achieve maximum profits. Algorithmic and Systematic Trading identify futures by analysing models, trends, patterns across days or months or years, in various forms of trading like high-frequency trading, day trading, low-frequency trading etc . This requires deep analysis of large historical and live streaming data, which comprises of variables like date, product type, stock symbol etc.

The analysis of such large data can't be achieved on a single system. It requires the use of a distributed system.

A distributed system is a model in which components located on networked computers communicate and coordinate their actions by passing messages. The components interact with each other in order to achieve a common goal.

This project aims to design and implement such a distributed system which can be effectively utilized for trade research analytics and back testing of trade strategies.

The Distributed system will utilize various big data technologies such as Spark for running queries on Esper engine for the purpose of back testing and research analytics. Instead of storing the data and running queries against stored data, the Esper engine allows applications to store queries and run the data through. Response from the Esper engine is real-time when conditions occur that match queries. Different team was working on the esper engine part. I worked on making this task distributed on various nodes in hadoop cluster using spark framework and thereby reducing the burden on single machine machine by reducing the time taken to done the backtesting.

Post Trade Analytics is one of the most important aspects of this project done which required technologies such as Elastic Search, Logstash and Kibana (popularly called the ELK Stack). After the strategies that have been run for Backtesting as well as on live market data for trading, log files are generated using the logger class of the log4j API, a reliable, fast and flexible logging

framework (APIs) written in Java. Moreover, various analysis is done everyday on distibuted framework using spark as well as by the various individual researchers on their systems which results in tremendous log files generated every day which can range from 100 GB to about 200 GB on last trading day of wek as well as month. Thus writing a java code using spark API’s is time consuming task for a process where data size can actually vary compared to the fix amount of market data received everyday. Thus we need a tool that makes this task simple for different varities of logs generated and works in a distributed manner.

Moreover, mining knowledge out of this varieties of data requires expertise and a large team, thus we need a tool that can transform plain logs to extract useful information out of it, store this data on a distributed cluster in a fast and effiecient manner which can be easily searched usign queriesas well as visualized so that it becomes easy for the non technical background employees such as traders because they are the ones who really need to see the results of the startegies used in their trades and do the post trade analytics to generate even better profits on the next trades they perform.

Keeping this scenario into account, ELK Stack is the most sought after technology and a distributed framework that helps us achieve an efficient way of doing post trade analytics.

**PROBLEM DEFINITION**

The aim of this project is to create a distributed system framework for back testing and post trade analytics. This project utilizes various technologies such as Spark, Elasticsearch, Logsatsh, Kibana and optimally run thousands of terabytes of data on thousands of commodity hardware nodes, producing results in the shortest amount of time possible and handle node failure. The problem can be broken down into 3 main stages mainly:

* Using the power of distributed processing engine Spark to reduce the time taken by Esper (Complex Event Processing Engine) for backtesting on live simulated market data.
* Post Trade Analytics (By using logs generated from the strategies (Java Programs) run on data:
  1. Using Logstash to transform logs (generated by running various strategies on market data) and using filebeat to ship logs from production server to testing server.
  2. Storing the useful data extracted from logstash in ElasticSearch, a NoSQL Database as well as an effiecient full text search engine where the meaningful log data generated by logstash as JSON events can be stored as well as searched with very very less latency.
  3. Using Kibana for visualizing the data stored in ElasticSearch where those with no knowledge about about programming/technology but tremendous knowledge on how the financial markets work can view the results and perform the post trade analytics.

**METHODOLOGY**

1. **Spark:**

Spark is a fast, in-memory data processing engine with elegant and expressive development APIs to allow data workers to efficiently execute streaming or SQL workloads that require fast iterative access to datasets.

Spark enables us to use distributed collection data-structure, known as Resilient distributed datasets (RDD), to make references to data, which can then be used in map-reduce jobs. Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes. There are two ways to create RDDs − parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Esper is is Complex Event Processing engine for performing EPL(Event Processing Language) queries, which are SQL like queries on streaming data for backtesting and analytics. But the issue is that it takes time for a single machine to process queries on huge incoming data and thus spark comes into picture, for performing this analytics in a distributed manner.

Major Spark API used for this purpose is **Spark SQL** which is a Spark component that supports querying data either via SQL or via the [Hive Query Language](https://cwiki.apache.org/confluence/display/Hive/LanguageManual). Spark SQL is a Spark module for structured data processing and is all about distributed in-memory computations on massive scale. It is the entry point for working with structured data (rows and columns) in Spark. Various functionalities of spark used are:

**import** **org.apache.spark.sql.SparkSession**

**val** spark **=** **SparkSession**

.builder()

.appName("Spark SQL basic example")

.config("spark.some.config.option", "some-value")

.getOrCreate()

val sqlDF = spark.sql("SELECT \* FROM people")

sqlDF.show()

OUTPUT => // | age| name|

// | 30| Andy|

// | 19| Justin|

* **Encoder** is the fundamental concept in the **serialization and deserialization (SerDe) framework** in Spark SQL 2.0. They are used to convert a JVM object of type T to and from the internal Spark SQL representation.
* **Row** is a generic row object with an ordered collection of fields that can be accessed by an [index](https://jaceklaskowski.gitbooks.io/mastering-apache-spark-2/spark-sql-Row.html#apply-index) or a name. Row belongs to org.apache.spark.sql.Row package. The traits of Row are -

length or size - Row knows the number of elements (columns).

schema - Row knows the schema [RowEncoder](https://jaceklaskowski.gitbooks.io/mastering-apache-spark-2/spark-sql-RowEncoder.html) takes care of assigning a schema to a Row

To create a new Row, use RowFactory.create() in Java or Row.apply() in Scala. RowEncoder is part of the Encoder Class and acts as a encoder for the DataFrames (or DataSets).

* **StructType** is a built-in [data type](https://jaceklaskowski.gitbooks.io/mastering-apache-spark-2/spark-sql-DataType.html) in Spark SQL to represent a collection of [StructField](https://jaceklaskowski.gitbooks.io/mastering-apache-spark-2/spark-sql-StructField.html)s (It has a name, the type and whether or not it be empty) that together define a schema or its part. A **schema** is the description of the structure of your data (which together create a [Dataset](https://jaceklaskowski.gitbooks.io/mastering-apache-spark-2/spark-sql-Dataset.html) in Spark SQL). StructType and StructField belong to the org.apache.spark.sql.types package.
* A **Dataset** is a distributed collection of data. Its is an evolution of RDDs in later versions of spark. Dataset is a interface that provides benefits of Spark RDDs as well as Spark SQL’s optimized execution engine. A Dataset can be [constructed](https://spark.apache.org/docs/2.1.0/sql-programming-guide.html#creating-datasets) from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.). Operations available on Datasets are divided into transformations and actions. Transformations are the ones that produce new Datasets, and actions are the ones that trigger computation and return results. Example transformations include map, mapPartitions, etc and Example actions include collect, show, or writing data out to file systems

A DataFrame (also known as Dataset of Rows) is a Dataset organized into named columns. It is conceptually equivalent to a table in a relational database, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of [sources](https://spark.apache.org/docs/2.1.0/sql-programming-guide.html#data-sources) such as: structured data files, tables in Hive, external databases, or existing RDDs. Dataset<Row> is used to represent a DataFrame.

To efficiently support domain-specific objects, an [Encoder](https://spark.apache.org/docs/2.1.0/api/scala/org/apache/spark/sql/Encoder.html) is required

* **SqlContext is** the entry point in spark for working with structured data (rows and columns) like data stored in MySQL or Hive.

**Pseudo Code:**

Encoder<Feed> feedEncoder = Encoders.bean(Feed.class);

Dataset<Feed> dataset = sparkSql.sql(sqlQuery).as(feedEncoder)

StructField[] sf = new StructField[sizeRow]

sf[0] = DataTypes.createStructField("day", DataTypes.DateType, false).......

ExpressionEncoder<Row> rowEncoder = RowEncoder.apply(new StructType(sf))

List<Row> = dataset.mapPartitions((MapPartitionsFunction<Feed, Row>) iterator -> {

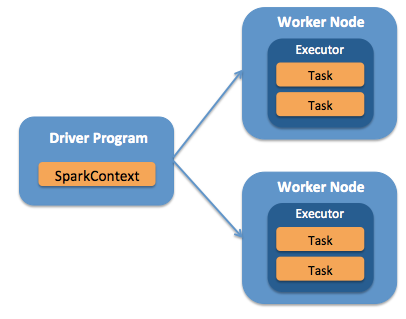
Esper Engine Instance created

The class called where order are placed and trades take place

Esper Instance assigned to every partition of data thus making Esper Process parallelized

return resultsStore.getResults()

}, rowEncoder).collectAsList()



* 1. **Spark:**

Spark is a fast, in-memory data processing engine with elegant and expressive development APIs to allow data workers to efficiently execute streaming, machine learning or SQL workloads that require fast iterative access to datasets. With Spark running on Apache Hadoop YARN, developers everywhere can now create applications to exploit Spark’s power, derive insights, and enrich their data science workloads within a single, shared dataset in Hadoop.

Spark enables us to use distributed collection data-structure, know as Resilient distributed datasets (RDD), to make references to data, which can then be used in map-reduce jobs. Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes. There are two ways to create RDDs − parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Traditionally the map-reduce jobs are slow on interactive and iterative queries, due to replication, serialization and disk I/O, RDD solves this bottle necks and speeds up the process by 90%. The difference of approach between traditional map-reduce approach and the approach through spark RDD have been illustrated in the below figures10 and 11

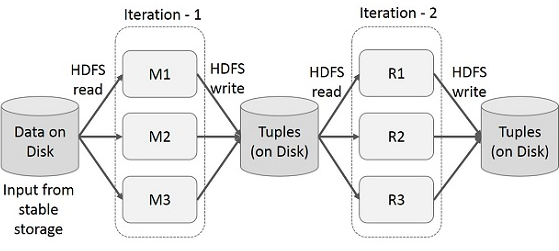


Figure 10

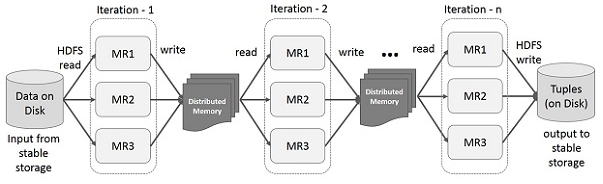


Figure 11

RDD enables to store the data on a Distributed Memory, instead of frequently calling the dataset, this distributed memory, it stores this state of memory as an object across the jobs and the object is sharable between those jobs. If the data cannot be stored in a Distributed data of the clusters ram, then the additional data is spilled on to the disk space. We can also persist the RDD in memory thus saving re-computation time.

* 1. The stages of an RDD can be broadly divided into two categories:
     1. Transformations.
     2. Actions.

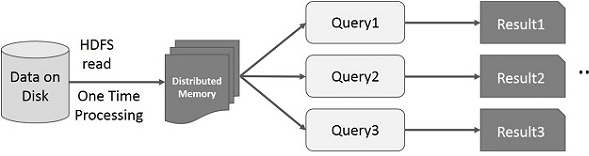
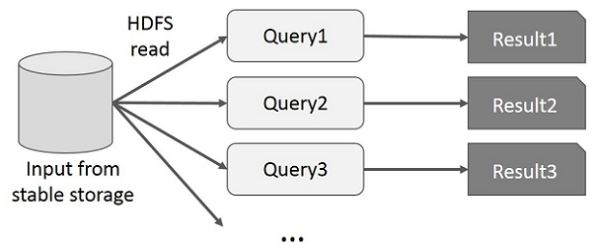


Figure 12

Transformations are the process that transform RDD from one form to another, for example if we have a RDD which is a un-partitioned dataset and we apply a process on it that transforms into a RDD with a Tuple, where in the tuple consists of pair corresponding to the data and a index, then we say that the RDD has been transformed from a un-partitioned RDD to a RDD of tupels.

RDD transformations returns pointer to new RDD and allow you to create dependencies between RDDs. Each RDD in dependency chain (String of Dependencies) has a function for calculating its data and has a pointer (dependency) to its parent RDD.

RDD transformation is not a set of data but is a step in a program (might be the only step) telling Spark how to get data and what to do with it.

Actions are the reduce part of the RDD, they are generally the last steps in a specific map-reduce RDD task, once spark reaches this stage, it completes the creation of plan and starts the execution. From the fig.13, we can see that there can be several number of transformations before an action takes place.

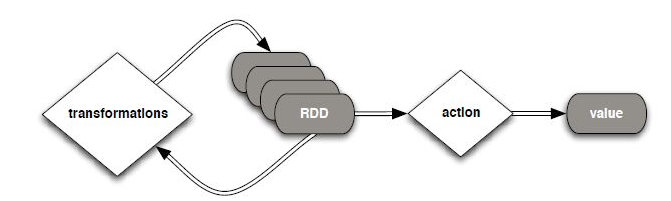


Figure 13

Spark is based on lazy programme principle, where in there is no execution unless an action is stated.

This means that the programme doesn’t start to execute as soon as it is started, the Spark first scans through the entire programme and then takes into account all the transformations that have been called, it then creates a plan of execution in form of a DAG. For, example for the following code snippet in fig<fig no:>, the resulting plan of execution has been shown in fig<fig no:>

The driver is the process that runs the user code that creates RDDs, and performs transformation and action, and also creates SparkContext. When the Spark Shell is launched, this signifies that we have created a driver program. On the termination of the driver, the application is finished.

The driver program splits the Spark application into the task and schedules them to run on the executor. The task scheduler resides in the driver and distributes task among workers. The two main key roles of drivers are:

* Converting user program into the task.
* Scheduling task on the executor.

The structure of Spark program at a higher level is: RDDs are created from some input data, derive new RDD from existing using various transformations, and then after it performs an action to compute data. In Spark Program, the DAG (directed acyclic graph) of operations are created implicitly. And when the driver runs, it converts that Spark DAG into a physical execution plan. The figure 14 shows the timeline view of the various tasks that the spark executes.

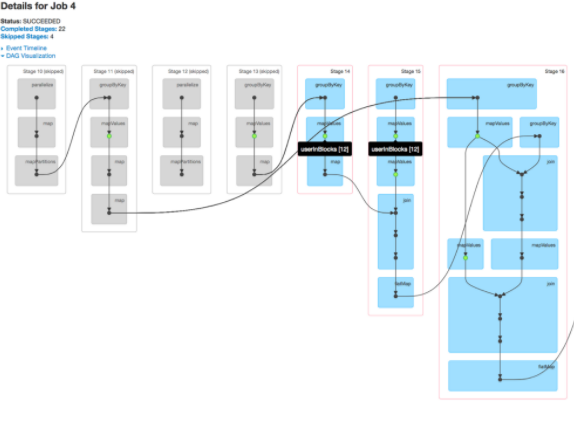


Figure 14

At high level, when any action is called on the RDD, Spark creates the DAG and submits it to the DAG scheduler. The DAG scheduler divides operators into stages of tasks. A stage is comprised of tasks based on partitions of the input data. The DAG scheduler pipelines operators together. For e.g. Many map operators can be scheduled in a single stage. The final result of a DAG scheduler is a set of stages.

The Stages are passed on to the Task Scheduler. The task scheduler launches tasks via cluster manager (Yarn/Standalone). The task scheduler doesn't know about dependencies of the stages.

The Worker executes the tasks on the Slave.

At high level, there are two transformations that can be applied onto the RDDs, namely narrow transformation and wide transformation. Wide transformations basically result in stage boundaries.

Narrow transformation - doesn't require the data to be shuffled across the partitions. for example, Map, filter etc..

wide transformation - requires the data to be shuffled for example, reduceByKey etc..

For example the following code snippet in fig 15, that counts the logs based on severity level is executed as follows :

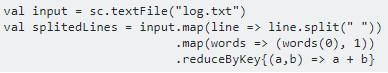


Figure 15

This sequence of commands implicitly defines a DAG of RDD objects (RDD lineage) that will be used later when an action is called. Each RDD maintains a pointer to one or more parents along with the metadata about what type of relationship it has with the parent. For example, when we call val b = a.map() on a RDD, the RDD b keeps a reference to its parent a, that's a lineage.

To display the lineage of an RDD, Spark provides a debug method toDebugString(). For example executing toDebugString() on the splitedLines RDD, will output the following, in fig 16.

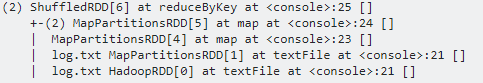


Figure 16

The first line (from the bottom) shows the input RDD. We created this RDD by calling sc.textFile().

Once the DAG is build, the Spark scheduler creates a physical execution plan. As mentioned above, the DAG scheduler splits the graph into multiple stages, the stages are created based on the transformations. The narrow transformations will be grouped (pipe-lined) together into a single stage. So for our example, Spark will create two stage execution as follows, fig 17.

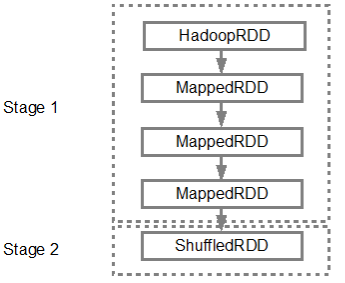


Figure 17

The DAG scheduler will then submit the stages into the task scheduler. The number of tasks submitted depends on the number of partitions present in the textFile. Fox example consider we have 4 partitions in this example, then there will be 4 set of tasks created and submitted in parallel provided there are enough slaves/cores. Below diagram illustrates this in more detail, fig 18:

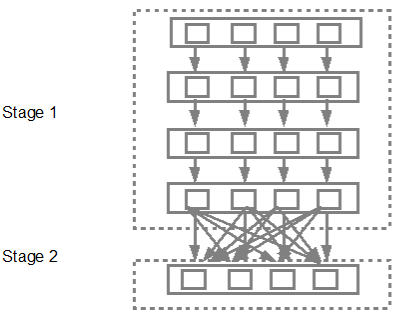


Figure 18

As shown in the above example spark can be used to design an composite map-reduce task, which can be used for various purposes.

The framework to be designed had to have the following functionalities:

* It has to use the data in the central data repository, earlier created and distribute it across various tasks
* Various simulations and strategies can be plugged in, thus creating a generic framework across all simulation and strategies.

The frame work was built using java and spark, initially the data was quired from within spark using hiveQL, then the data was transformed into a RDD of a custom class type, where in the class had the required data representation.

This was the raw data across the required days, this data was then transformed into a RDD of type tuple. Where in, the tuple represented was a pair of key/value, the key begin the timestamp of the data and the value begin the data itself. This transformed data was then mapped into a javaRDD Pair, for this a custom map partition function was developed.

On this RDD various combinations are to be run, to simulate a random data inflow. Each such simulation depends only on the data it takes in and has no other dependency on any other data, so each such data could be mapped as a separate process.

The data from the central data base was distributed into various RDD, based on the test cases and parameters, the test cases where generated through simulations, on which the multiple strategies where run across multiple test cases in parallel.

Traditionally, if there are n number of simulations to be run, the base code of simulation would be run in a loop for n times , changing the parameters of simulation in each iteration. The aim was to achieve this through parallelism rather than iterations.

When the driver initiates a map-job, the map-job is distributed into various clusters(slave nodes), each such executer then assigns various tasks, which are then run parallel, if the number of clusters are smaller than the number of executors, then each cluster is assigned more than one executer.

This can be further illustrated by fig 19.

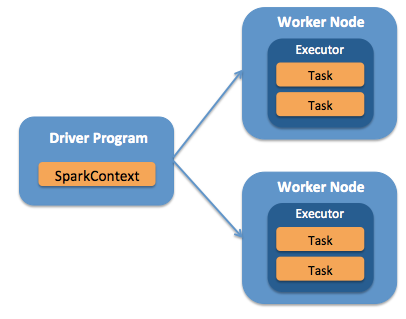


Figure 19

The problem with this approach was, that the executors under a cluster aka worker node, can only performs a single task, as the resources are assigned to the executor, which it then distributes to the tasks, iteratively.

Say, for example we have to run 10 simulations across 3 clusters, with 5 combinations of parameters P1, P2, P3, P4 and P5. When the map-reduce gets executed, let’s say 4 executors were created, now each cluster gets one executor and one cluster gets two executors, under each executor there are a number of tasks, each task inside an executor has to wait till the task before it completes.

This, leads to considerable delay in processing, it was required to speed up this process. To achieve massive parallelization, the tasks inside an executor that where running sequentially had to parallelized.

To achieve this, we introduced the concept of resource sharing inside the executor of a cluster.

Each executor is given a set of resources, which are utilized by the task, in sequential order, these resources are to be put in a common resource sharing paradigm, where in all the tasks inside a executor, can access them parallel.

This, way of sharing resources ensures that, the tasks are not run sequentially, but can be run parallel. This will lead to greater reduction in the time taken to achieve the tasks. This has been illustrated in the fig,<fig No:><to be added> . This parallelization, was achieved by multi-threading the tasks inside the executor, the performance increments achieved, have been illustrated in the fig,<fig No:><to be added>

The next bottle neck, was the I/O calls required to access the data, this was taking a considerable time, as the number of combinations, required for the simulations increased, the number of I/O calls Increased, leading to increase in the time required to run the simulations.

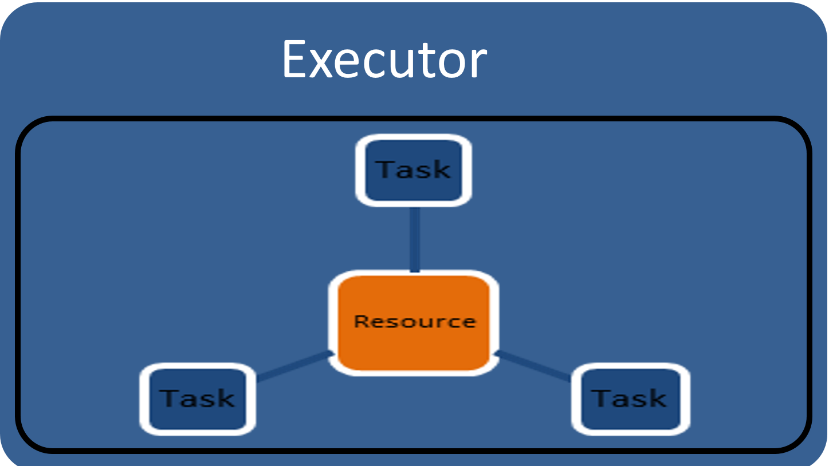


Figure 20

If the parameters of the simulation are same, but the number of combinations of those parameters keep changing, then the data acquired for one simulation, could be used across all the simulations with similar parameters. Hence, a cache layer was created, between the central database and framework, where in the parametric data common across a set of simulations is brought into.

The cache layer was dynamically divided into different partitions, depending upon the set of simulations being run in parallel. Thus, the number of I/O, for n combinations of p parameters, has been reduce from n to 1, as only once the data had to be fetched and cached. If the data is larger than the cache memory, then the data gets spilled on to the disk.

Using all the above mentioned principles and changes to the existing system, an robust, optimal back testing framework was built. The data generated from these simulations was then stored in the central database created partitioned by the date the simulation run, the researcher id of the simulation and was stored in map-columnar format, as mentioned in <Index No:><to be added>

The entire workflow and design of algorithm has been illustrated in the fig<fig No:><to be added>

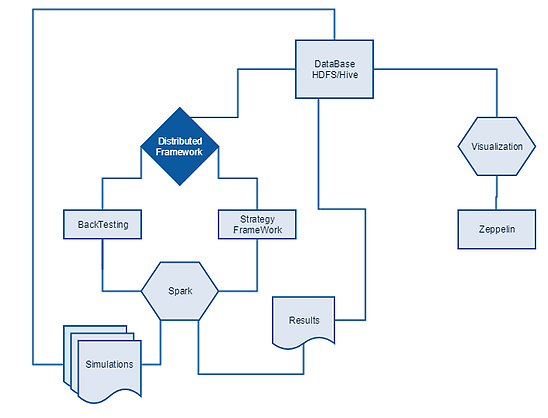


Figure 21

Thus, the entire framework for back-testing and analytics was built.

**2) Post Trade Analytics**

### Why is Log Analysis Becoming More Important?

As money at Edelweiss Global Markets is earned by putting money in markets, thus verifying logs generated after strategy (Java program on single machine or distributed cluster) is run and analyzing these logs is becoming more and more critical.

In cloud-based infrastructures or distributed infrastructure, performance isolation is extremely difficult to reach — particularly whenever systems are heavily loaded. The performance of virtual machines in the cloud can greatly fluctuate based on the specific loads, infrastructure servers, environments, and number of active users. As a result, reliability and node failures can become significant problems. Log management platforms can monitor all of these infrastructure issues as well as process operating system logs.

### The Challenges of Disparate Logs Data?

**No Consistency —**The variety of systems and absence of standards means that it's difficult to be a jack-of-all trades.

* Logging is different for each app, system, or device
* Specific knowledge is necessary for interpreting various types of logs
* Variation in format makes it challenging to search
* Many types of time formats

**No centralization —** Simply put, log data is everywhere:

* Logs in many locations on various servers
* Many locations of various logs on each server

**Accessibility of Log Data —** Much of the data is difficult to locate and manage. Although some of the log data may be highly valuable, many admins face these steep challenges:

* Access is often difficult
* High expertise to mine data
* Logs can be difficult to find
* Immense size of Log Data

The ELK stack helped us manage each of these challenges, and more. ELK is best for time-series data-anything with a time stamp-such as you'll find in most web server logs, transaction logs, and stock data listings. To be intelligible, these logs usually need substantial clean-up.

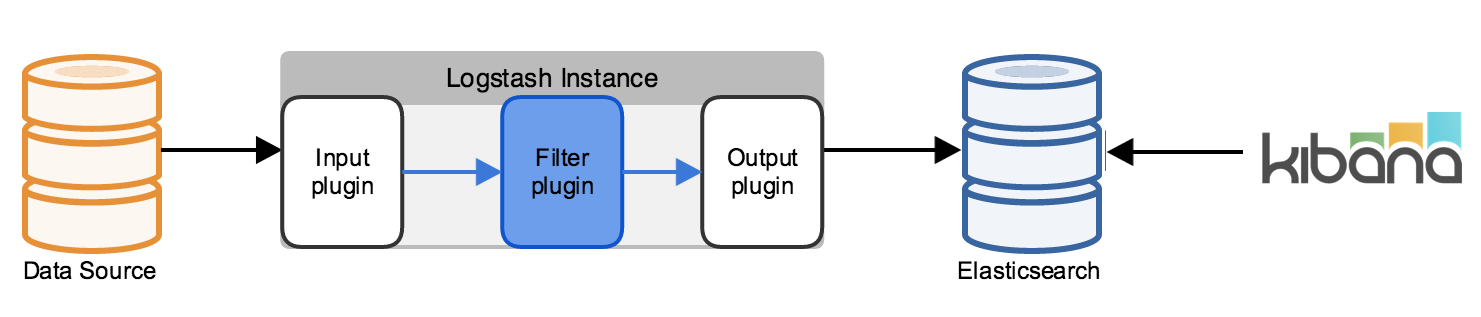
**What is the ELK Stack?**

The ELK Stack is a collection of three open-source products — [Elasticsearch](https://logz.io/category/blog/elasticsearch/), [Logstash](https://logz.io/category/blog/logstash/), and [Kibana](https://logz.io/category/blog/kibana/)— from [Elastic](https://www.elastic.co/). Elasticsearch is a NoSQL database that is based on the Lucene search engine. Logstash is a log pipeline tool that accepts inputs from various sources, executes different transformations, and exports the data to various targets. Kibana is a visualization layer that works on top of Elasticsearch.

Together, these three different open source products are most commonly used in log analysis in IT environments (though there are many more use cases for the ELK Stack starting including business intelligence, security and compliance, and web analytics). Logstash collects and parses logs, and then Elasticsearch indexes and stores the information. Kibana then presents the data in visualizations that provide actionable insights into one’s environment.

To sum up, ELK stack setup has three main components:

* **Logstash**: The server component of Logstash that processes incoming logs
* **Elasticsearch**: Stores all of the logs
* **Kibana**: Web interface for searching and visualizing logs



### Why is ELK So Popular?

The ELK Stack is popular because it fulfills a need in the log analytics space. Splunk’s enterprise software has long been the market leader, but its numerous functionalities are increasingly not worth the expensive price.

After all, top tech companies of the world like Netflix, Facebook, Microsoft, LinkedIn, and Cisco monitor their logs using the ELK Stack.

Strategy Logs are of various types, but the common part of all is the Order Placed information in long or error generated. Let’s take an example of sample log data generated:

15:29:19,116 Level[INFO] [pool-6-thread-2] ## New Order=> Portfolio = JUNE9900P;Security = NIFTY17JUN8200PE::NSE\_FO;Order ID = 61686479;Account =2225;Side =SELL;RejectionReason: null;Quantity = 750;NewQuantity = 750;Price = 0.15;NewPrice = 0.15;TriggerPrice = 0.0;NewTriggerPrice = 0.0;OrderStatus = NOT\_SENT;NewOrderStatus = NOT\_SENT;OrderType = IOC;NewOrderType = IOC;FilledQuantity = 0;AverageFillPrice = 0.0;ExpectedPrice = 0.15;Bid = 0.15;LTP = 0.15;Ask = 0.2;ParentOrderID=0;ExchangeOrderID=0;OriginalExchangeOrderID=0;stsnId=9111074429831258055;Comment= ## com.edelweiss.algo.portfolio.UnsynchronizedDefaultPortfolioSecurity.placeOrder(UnsynchronizedDefaultPortfolioSecurity.java:494)

09:15:26,662 Level[ERROR] [pool-6-thread-2] ## Bid/Ask of securities is not set for constituent ID 3 NIFTY17AUG9900PE::NSE\_FO AUG8000C ## com.edelweiss.algo.arb.ml.fourLeg.strategy.FourLegStrategy.priceUpdate(FourLegStrategy.java:144)

## Elasticsearch

Elasticsearch is a highly scalable open-source full-text search and analytics engine. It allows you to store, search, and analyze big volumes of data quickly and in near real time. It is generally used as the underlying engine/technology that powers applications that have complex search features and requirements.

Elasticsearch is a NoSQL database that is based on the Lucene search engine.. That means it stores data in an unstructured way and that you cannot use SQL to query it. Unlike most NoSQL databases, though, Elasticsearch has a strong focus on search capabilities and features — so much so, in fact, that the easiest way to get data from Elasticsearch is to search for it using the REST API.

**Key Features of ElasticSearch:**

* **Real-time data and real-time analytics.** The ELK stack gives us the power of real-time data insights, with the ability to perform super-fast data extractions from virtually all structured or unstructured data sources. Real-time extraction, and real-time analytics. Elasticsearch is the engine that gives you both the power and the speed.
* **Scalable, high-availability, multi-tenant.** With Elasticsearch, you can start small and expand it along with your business growth-when you are ready. It is built to scale horizontally out of the box. As you need more capacity, simply add another node and let the cluster reorganize itself to accommodate and exploit the extra hardware. Elasticsearch clusters are resilient, since they automatically detect and remove node failures. You can set up multiple indices and query each of them independently or in combination.
* **Full text search.**Under the cover, Elasticsearch uses Lucene to provide the most powerful full-text search capabilities available in any open-source product. The search features come with multi-language support, an extensive query language, geolocation support, and context-sensitive suggestions, and autocompletion.
* **Document orientation.** You can store complex, real-world entities in Elasticsearch as structured JSON documents. All fields have a default index, and you can use all the indices in a single query to get precise results in the blink of an eye.

## Basic Concepts

There are a few concepts that are core to Elasticsearch

* **Near Real Time:** Elasticsearch is a near real time search platform. What this means is there is a slight latency (normally one second) from the time you index a document until the time it becomes searchable.
* **Node** is a running instance of elasticsearch which belongs to a [cluster](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-cluster). Multiple nodes can be started on a single server for testing purposes, but usually you should have one node per server. At startup, a node will use unicast to discover an existing cluster with the same cluster name and will try to join that cluster.
* **Cluster** consists of one or more [nodes](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-node) which share the same cluster name. Each cluster has a single master node which is chosen automatically (or we can specify in the elasticsearch conf.yml file of on each node to specify which nodes to chose as the master nodes) by the cluster and which can be replaced if the current master node fails.
* **Document** is a JSON document which is stored in elasticsearch. It is like a row in a table in a relational database. Each document is stored in an [index](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-index) and has a [type](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-type) and an [id](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-id). A document is a JSON object (also known in other languages as a hash / hashmap / associative array) which contains zero or more [fields](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-field), or key-value pairs. The original JSON document that is indexed will be stored in the [\_source field](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-source_field), which is returned by default when getting or searching for a document.
* **Shards**  is a single Lucene instance. It is a low-level “worker” unit which is managed automatically by elasticsearch. An index is a logical namespace which points to [primary](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-primary-shard) and [replica](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-replica-shard) shards. Other than defining the number of primary and replica shards that an index should have, you never need to refer to shards directly. Instead, our code should deal only with an index. Elasticsearch distributes shards amongst all [nodes](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-node) in the [cluster](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-cluster) and our data in index is distributed across shards, and can move shards automatically from one node to another in the case of node failure, or the addition of new nodes.

Each document is stored in a single primary [shard](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-shard). When you index a document, it is indexed first on the primary shard, then on all [replicas](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-replica-shard) of the primary shard. By default, an [index](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-index) has 5 primary shards. You can specify fewer or more primary shards to scale the number of [documents](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-document) that your index can handle. You cannot change the number of primary shards in an index, once the index is created.

* **Index** is like a table in a relational database. It is a collection of documents that have somewhat similar characteristics. For example, if we have an index for customer data, another index for a product catalog, and yet another index for order data. An index is identified by a name (that must be all lowercase) and this name is used to refer to the index when performing indexing, search, update, and delete operations against the documents in it. In a single cluster, we can define as many indexes as you want.
* **Type** represents the type of document. Within an index, you can define one or more types. A type is a logical category/partition of your index whose semantics is completely up to you. In general, a type is defined for documents that have a set of common fields. Thus, the search API can filter documents by type.
* **ID** of a [document](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-document) identifies a document. The index/type/id of a document must be unique. If no ID is provided, then it will be auto-generated.
* **Field:** A [document](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-document) contains a list of fields, or key-value pairs. The value can be a simple (scalar) value (eg a string, integer, date), or a nested structure like an array or an object. A field is similar to a column in a table in a relational database. The [mapping](https://www.elastic.co/guide/en/elasticsearch/reference/5.1/glossary.html#glossary-mapping) for each field has a field type that indicates the type of data that can be stored in that field, eg integer, string, object. The mapping also allows you to define (amongst other things) how the value for a field should be analyzed.

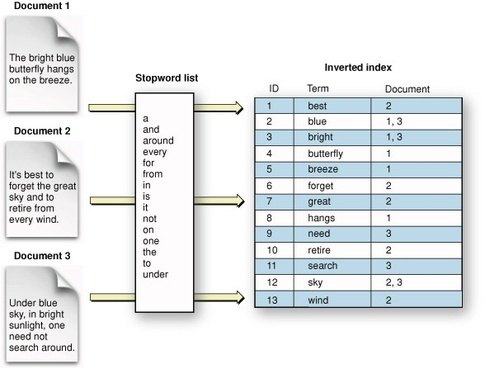
**How ElasticSearch stores Data**

In the real world, though, not all entities of the same type look the same. One person might have a home telephone number, while another person has only a cell-phone number, and another might have both. One of the reasons that object-oriented programming languages are so popular is that objects help us represent and manipulate real-world entities with potentially complex data structures.

The problem comes when we need to store these entities. Traditionally, we have stored our data in columns and rows in a relational database, the equivalent of using a spreadsheet. All the flexibility gained from using objects is lost because of the inflexibility of our storage medium. But what if we could store our objects as objects? Instead of modeling our application around the limitations of spreadsheets, we can instead focus on *using* the data. The flexibility of objects is returned to us.

An *object* is a language-specific, in-memory data structure. To send it across the network or store it, we need to be able to represent it in some standard format. [JSON](http://en.wikipedia.org/wiki/Json) is a way of representing objects in human-readable text. It has become the de facto standard for exchanging data in the NoSQL world. When an object has been serialized into JSON, it is known as a *JSON document*. Elasticsearch is a distributed *document* store. It can store and retrieve complex data structures—serialized as JSON documents—in *real time*. In other words, as soon as a document has been stored in Elasticsearch, it can be retrieved from any node in the cluster.

Of course, we don’t need to only store data; we must also query it and at speed. While NoSQL solutions exist that allow us to store objects as documents, they still require us to think about how we want to query our data, and which fields require an index in order to make data retrieval fast. In Elasticsearch, *all data in every field* is *indexed by default*. That is, every field has a dedicated inverted index for fast retrieval. And, unlike most other databases, it can use all of those inverted indices *in the same query*, to return results at breathtaking speed



**The Concept of Sharding**

Sharding is important for two primary reasons:

* It allows you to horizontally split/scale your content volume
* It allows you to distribute and parallelize operations across shards (potentially on multiple nodes) thus increasing performance/throughput

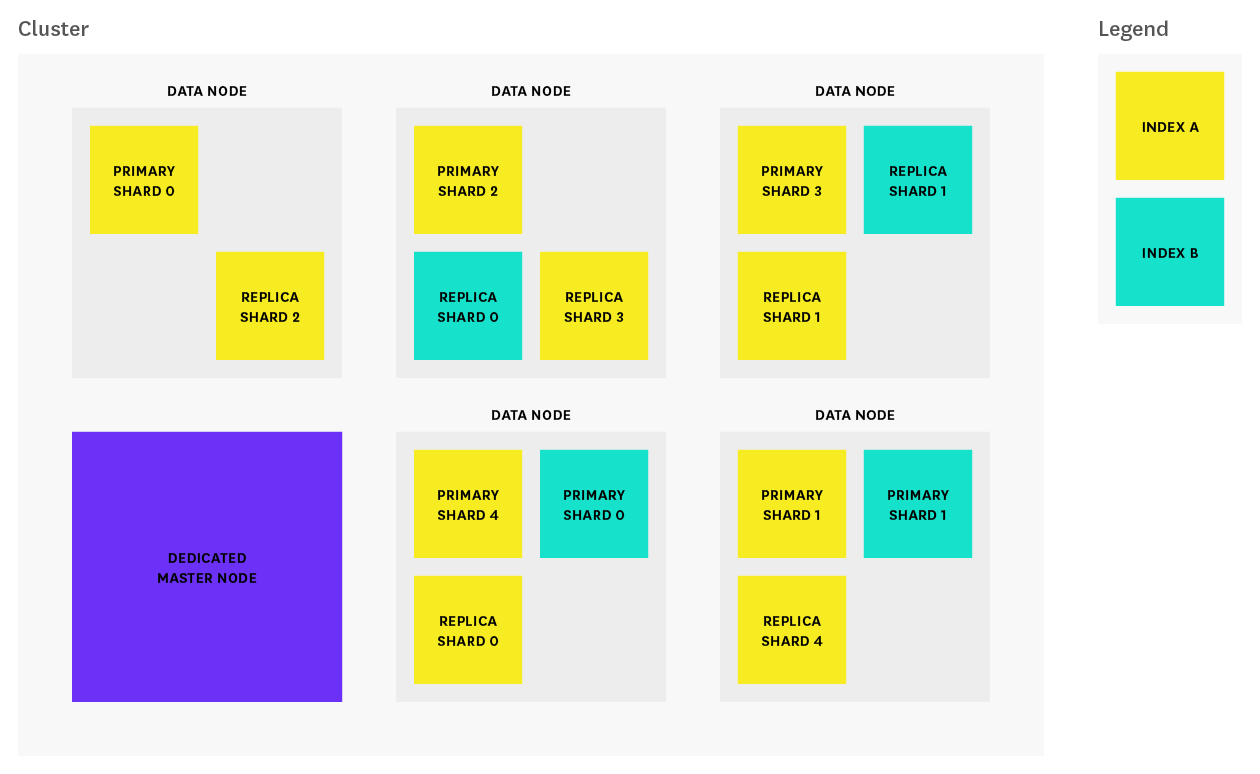
The mechanics of how a shard is distributed and also how its documents are aggregated back into search requests are completely managed by Elasticsearch and is transparent to you as the user.

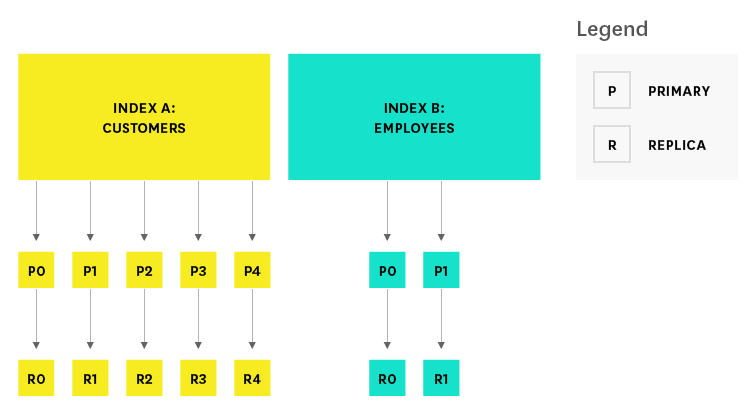
In a network/cloud environment where failures can be expected anytime, it is very useful and highly recommended to have a failover mechanism in case a shard/node somehow goes offline or disappears for whatever reason. To this end, Elasticsearch allows you to make one or more copies of your index’s shards into what are called replica shards, or replicas for short.

Replication is important for two primary reasons:

* It provides high availability in case a shard/node fails. For this reason, it is important to note that a replica shard is never allocated on the same node as the original/primary shard that it was copied from.
* It allows you to scale out your search volume/throughput since searches can be executed on all replicas in parallel.

Each Elasticsearch shard is a Lucene index. There is a maximum number of documents you can have in a single Lucene index. As of [LUCENE-5843](https://issues.apache.org/jira/browse/LUCENE-5843), the limit is 2,147,483,519 (= Integer.MAX\_VALUE - 128) documents. You can monitor shard sizes using the [\_cat/shards](https://www.elastic.co/guide/en/elasticsearch/reference/current/cat-shards.html) api.





To summarize, each index can be split into multiple shards. An index can also be replicated zero (meaning no replicas) or more times. Once replicated, each index will have primary shards (the original shards that were replicated from) and replica shards (the copies of the primary shards). The number of shards and replicas can be defined per index at the time the index is created. After the index is created, we can change the number of replicas dynamically anytime but you cannot change the number of shards after-the-fact.

By default, each index in Elasticsearch is allocated 5 primary shards and 1 replica which means that if you have at least two nodes in your cluster, your index will have 5 primary shards and another 5 replica shards (1 complete replica) for a total of 10 shards per index.

**Logstash**

A great use for the ELK Stack is the storing, visualization, and analysis of logs and other time-series data. Logstash is an integral part of the data workflow from the source to Elasticsearch and further. Not only does it allow us to pull data from a wide variety of sources, it also gives us the tools to filter, massage, and shape the data so that it’s easier to work with.

**Why is Logstash so popular?**

* **The ingestion workhorse for Elasticsearch and more**

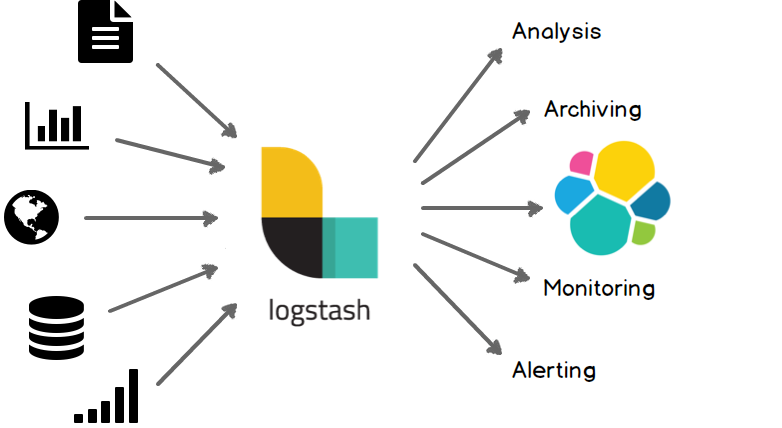
Horizontally scalable data processing pipeline with strong Elasticsearch and Kibana.

* **Pluggable pipeline architecture**

Mix, match, and orchestrate different inputs, filters, and outputs to play in pipeline harmony

* **Community-extensible and developer-friendly plugin ecosystem**

Over 200 plugins available, which make it easy for it to be used with any input and output



**Basic Logstash Concepts**

# Event: A single unit of information, containing a timestamp plus additional data. An event arrives via an input, and is subsequently parsed, timestamped, and passed through the Logstash [pipeline](https://www.elastic.co/guide/en/logstash/current/glossary.html#glossary-pipeline).

* **Pipeline:** A term used to describe the flow of [events](https://www.elastic.co/guide/en/logstash/current/glossary.html#glossary-event) through the Logstash workflow. A pipeline typically consists of a series of input, filter, and output stages. [Input](https://www.elastic.co/guide/en/logstash/current/glossary.html#glossary-input-plugin) stages get data from a source and generate events,[filter](https://www.elastic.co/guide/en/logstash/current/glossary.html" \l "glossary-filter-plugin) stages, which are optional, modify the event data, and [output](https://www.elastic.co/guide/en/logstash/current/glossary.html#glossary-output-plugin) stages write the data to a destination.
* **Indexer:** A Logstash instance that is tasked with interfacing with an Elasticsearch cluster in order to index [event](https://www.elastic.co/guide/en/logstash/current/glossary.html#glossary-event) data.

### Plugins: A self-contained software package that implements one of the stages in the Logstash event processing [pipeline](https://www.elastic.co/guide/en/logstash/5.x/glossary.html#glossary-pipeline). The list of available plugins includes [input plugins](https://www.elastic.co/guide/en/logstash/5.x/glossary.html#glossary-input-plugin), [output plugins](https://www.elastic.co/guide/en/logstash/5.x/glossary.html#glossary-output-plugin), [codec plugins](https://www.elastic.co/guide/en/logstash/5.x/glossary.html#glossary-codec-plugin), and [filter plugins](https://www.elastic.co/guide/en/logstash/5.x/glossary.html#glossary-filter-plugin). The plugins are implemented as Ruby [gems](https://www.elastic.co/guide/en/logstash/5.x/glossary.html#glossary-gem) and hosted on [RubyGems.org](https://rubygems.org/). You define the stages of an event processing [pipeline](https://www.elastic.co/guide/en/logstash/5.x/glossary.html#glossary-pipeline) by configuring plugins.

* **Shipper:** An instance of Logstash that send events to logstash or any other output
* **Worker:** The filter thread model used by Logstash, where each worker receives an event and applies all filters, in order, before emitting the event to the output queue. This allows scalability across CPUs because many filters are CPU intensive.

**How Logstash Works**

The Logstash event processing pipeline has three stages: inputs → filters → outputs. Inputs generate events, filters modify them, and outputs ship them elsewhere. Inputs and outputs support codecs that enable you to encode or decode the data as it enters or exits the pipeline without having to use a separate filter. The transformation pipeline of logstash workds using the help of following plugins.

### Input plugins

We use inputs to get data into Logstash. Some of the more commonly-used inputs are:

* **file**: reads from a file on the filesystem
* **syslog**: listens on the well-known port 514 for syslog messages
* **beats**: processes events sent by [Filebeat](https://www.elastic.co/downloads/beats/filebeat" \t "_top).

### Filter plugins

Filters are intermediary processing devices in the Logstash pipeline. You can combine filters with conditionals to perform an action on an event if it meets certain criteria. Some useful filters include:

* **grok**: parse and structure arbitrary text. Grok is currently the best way in Logstash to parse unstructured log data into something structured and queryable. With 120 patterns built-in to Logstash, it’s more than likely you’ll find one that meets your needs!
* **mutate**: perform general transformations on event fields. You can rename, remove, replace, and modify fields in your events.
* **drop**: drop an event completely, for example, debug events.
* **clone**: make a copy of an event, possibly adding or removing fields.
* **geoip**: add information about geographical location of IP addresses (also displays amazing charts in Kibana!)

**Output plugins**

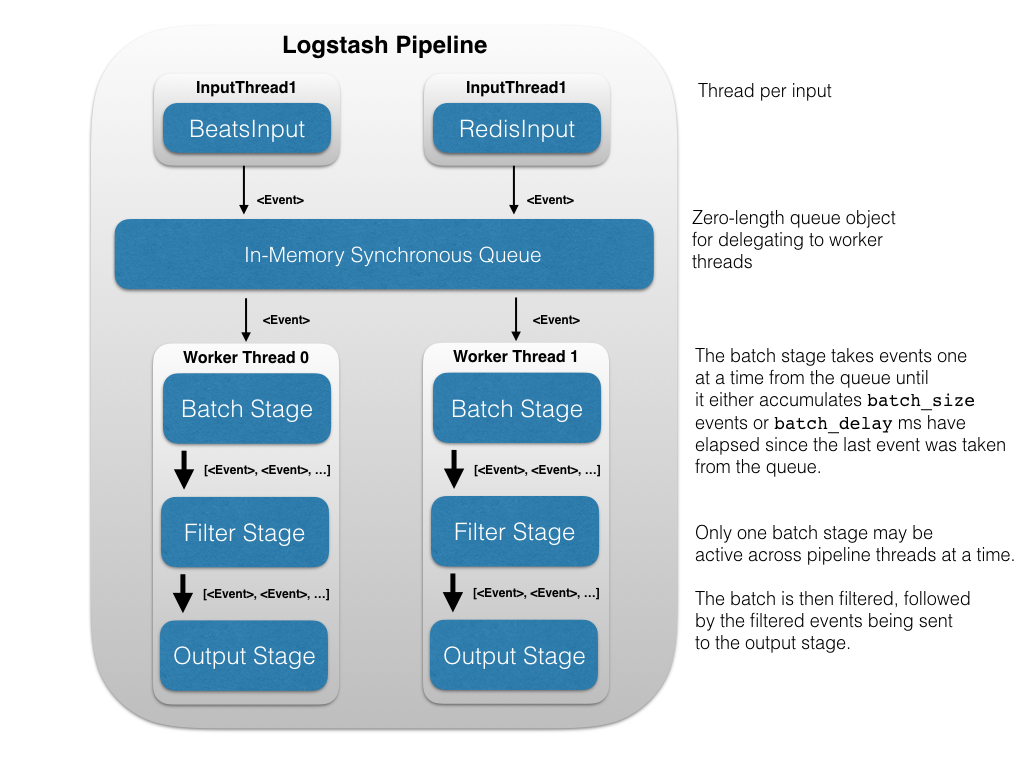
Outputs are the final phase of the Logstash pipeline. An event can pass through multiple outputs, but once all output processing is complete, the event has finished its execution. Some commonly used outputs include:

* **elasticsearch**: send event data to Elasticsearch. If we want to save our data in an efficient, convenient, and easily queryable format. Elasticsearch is the way to go.
* **file**: write event data to a file on disk.
* **stdout**: print event data directly on console

### Codec plugins

Codecs are basically stream filters that can operate as part of an input or output. Codecs enable you to easily separate the transport of your messages from the serialization process. Popular codecs include:

* **json**: encode or decode data in the JSON format.
* **multiline**: merge multiple-line text events such as java exception and stacktrace messages into a single event.



Rather than normalizing with time-sucking ETL (Extract, Transform, and Load), we recommend that you switch over to the fast track. Instead, you could spend much less time training Logstashto normalize the data, getting Elasticsearch to process the data, and then visualize it with Kibana. With Logstash, it's super easy to take all those logs and store them in a central location. The only prerequisite is a Java runtime, and it takes just two commands to get Logstash up and running.

Using Elasticsearch as a backend datastore and Kibana as a frontend dashboard (see below), Logstash will serve as the workhorse for storage, querying and analysis of your logs. Since it has an arsenal of ready-made inputs, filters, codecs, and outputs, you can grab hold of a very powerful feature-set with a very little effort on your part.

Think of Logstash as a pipeline for event processing: it takes precious little time to choose the inputs, configure the filters, and extract the **relevant, high-value** data from your logs. Take a few more steps, make it available to Elasticsearch and—BAM!—you get super-fast queries against your mountains of data.

## Execution Model[edit](https://github.com/elastic/logstash/edit/5.x/docs/static/life-of-an-event.asciidoc)

The Logstash event processing pipeline coordinates the execution of inputs, filters, and outputs.

Each input stage in the Logstash pipeline runs in its own thread. Inputs write events to a common Java [SynchronousQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/SynchronousQueue.html" \t "_top). This queue holds no events, instead transferring each pushed event to a free worker, blocking if all workers are busy. Each pipeline worker thread takes a batch of events off this queue, creating a buffer per worker, runs the batch of events through the configured filters, then runs the filtered events through any outputs. The size of the batch and number of pipeline worker threads are configurable (see [Tuning and Profiling Logstash Performance](https://www.elastic.co/guide/en/logstash/current/tuning-logstash.html)).

By default, Logstash uses in-memory bounded queues between pipeline stages (input → filter and filter → output) to buffer events. If Logstash terminates unsafely, any events that are stored in memory will be lost. To prevent data loss, you can enable Logstash to persist in-flight events to disk. See [Persistent Queues](https://www.elastic.co/guide/en/logstash/current/persistent-queues.html) for more information.

# Kibana

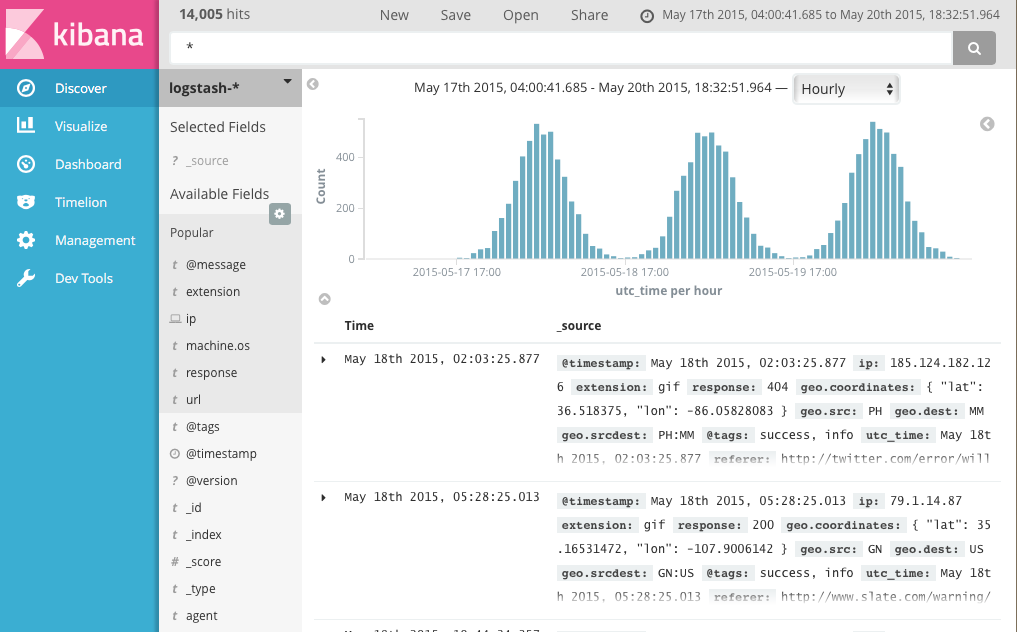
Kibana is an open source analytics and visualization platform designed to work with Elasticsearch. You use Kibana to search, view, and interact with data stored in Elasticsearch indices. You can easily perform advanced data analysis and visualize your data in a variety of charts, tables, and maps.

Kibana makes it easy to understand large volumes of data. Its simple, browser-based interface enables you to quickly create and share dynamic dashboards that display changes to Elasticsearch queries in real time.

Setting up Kibana is a snap. You can install Kibana and start exploring your Elasticsearch indices in minutes — no code, no additional infrastructure required.

Discovering Your Data[edit](https://github.com/elastic/kibana/edit/5.5/docs/getting-started/tutorial-discovering.asciidoc)

Click **Discover** in the side navigation to display Kibana’s data discovery functions:



In the query bar, you can enter an [Elasticsearch query](https://www.elastic.co/guide/en/elasticsearch/reference/5.x/query-dsl-query-string-query.html" \l "query-string-syntax" \t "_top) to search your data. You can explore the results in Discover and create visualizations of saved searches in Visualize.

The current index pattern is displayed beneath the query bar. The index pattern determines which indices are searched when you submit a query. To search a different set of indices, select different pattern from the drop down menu. To add an index pattern, go to **Management/Kibana/Index Patterns** and click **Add New**.

You can construct searches by using the field names and the values you’re interested in. With numeric fields you can use comparison operators such as greater than (>), less than (<), or equals (=). You can link elements with the logical operators AND, OR, and NOT, all in uppercase.

To try it out, select the ba\* index pattern and enter the following query string in the query bar:

account\_number:<100 AND balance:>47500

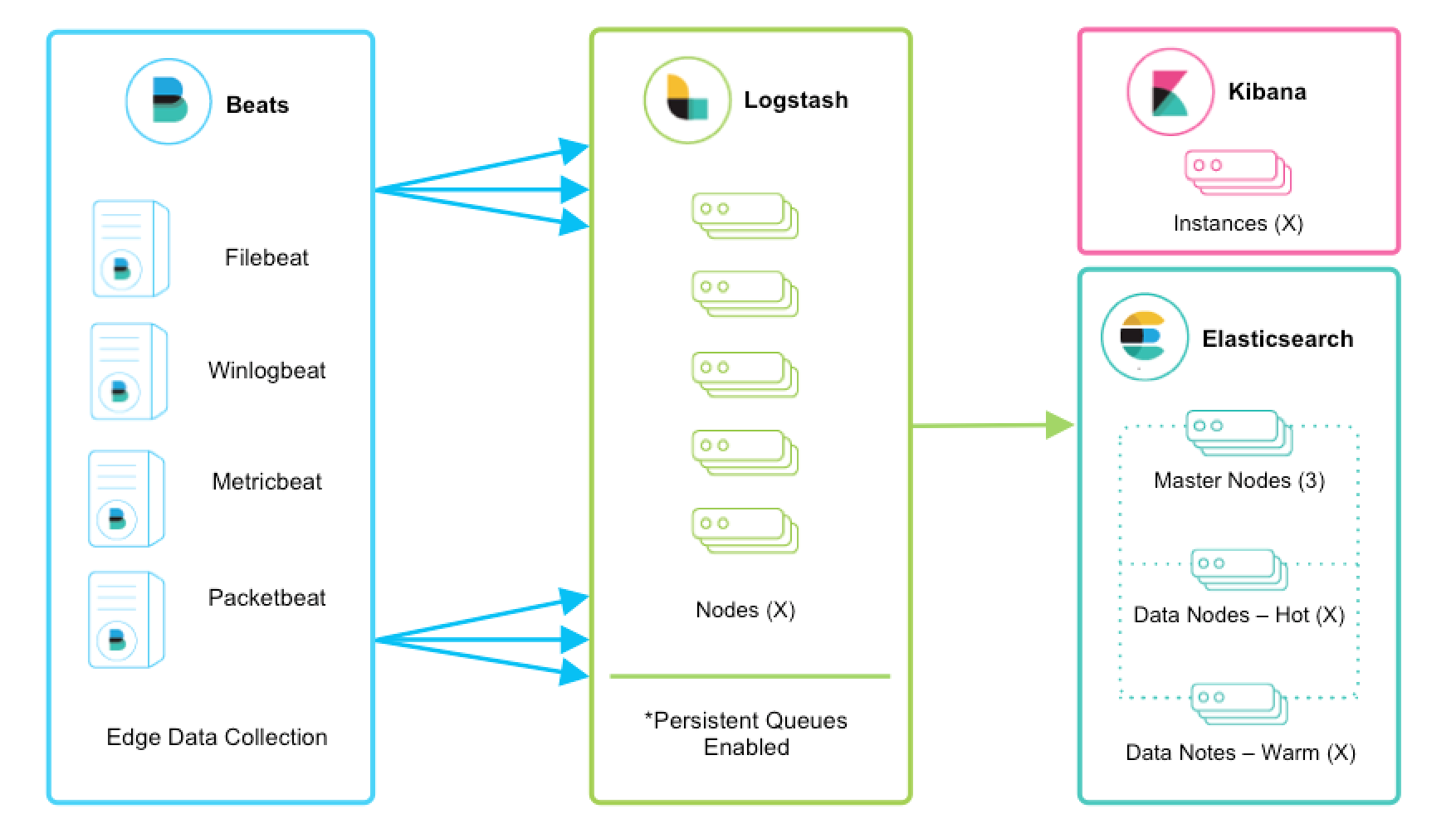
This query returns all account numbers between zero and 99 with balances in excess of 47,500. When searching the sample bank data, it returns 5 results: Account numbers 8, 32, 78, 85, and 97.

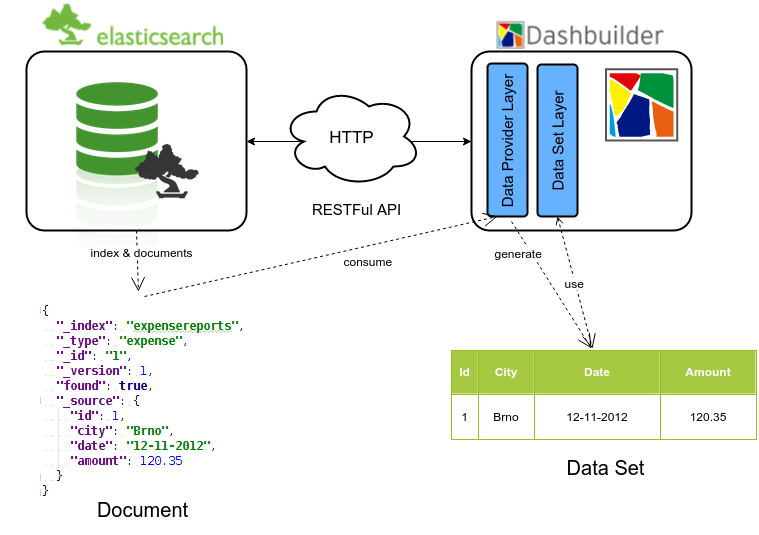
## Visualizing Your Data[edit](https://github.com/elastic/kibana/edit/5.5/docs/getting-started/tutorial-visualizing.asciidoc)

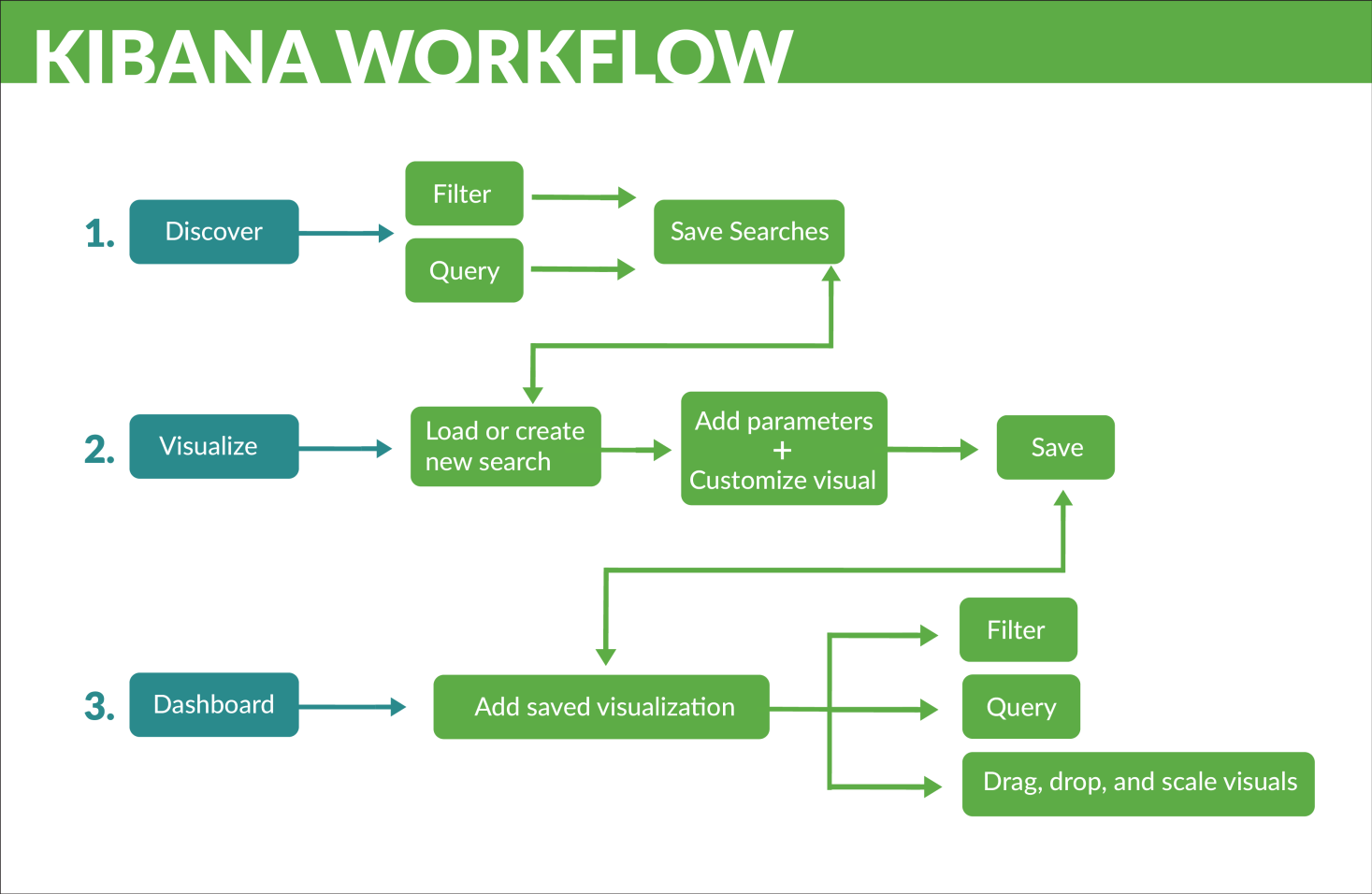
To start visualizing your data, click **Visualize** in the side navigation.

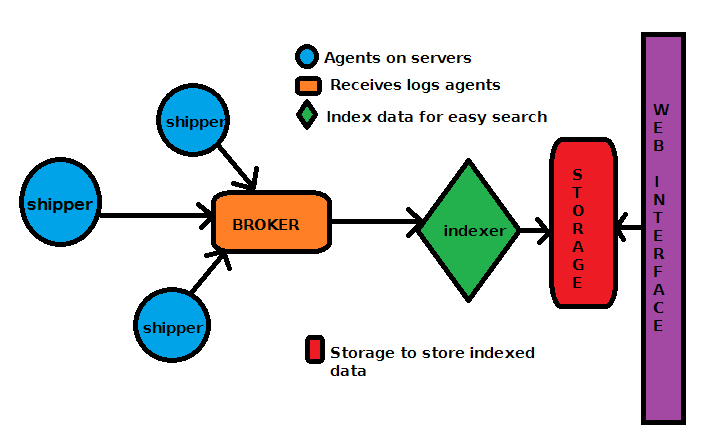
The **Visualize** tools enable you to view your data in several ways. For example, let’s use that venerable visualization, the pie chart, to get some insight into the account balances in the sample bank account data. To get started, click the big blue **Create a visualization** button in the center of the screen.

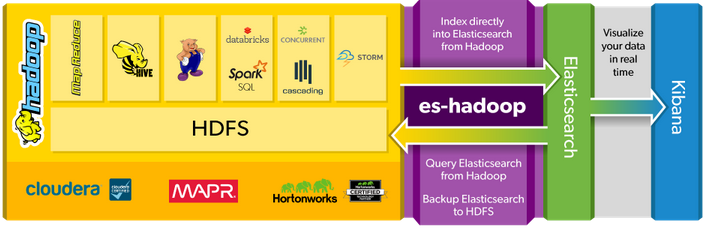
When you define an index pattern, indices that match that pattern must exist in Elasticsearch. Those indices must contain data.

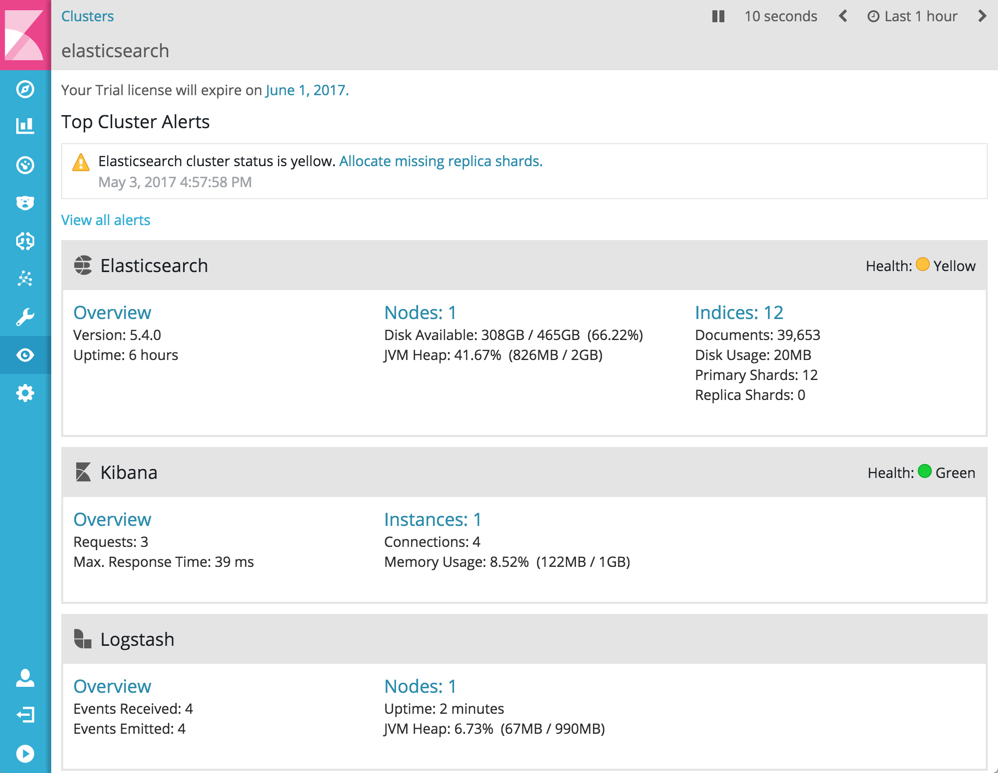


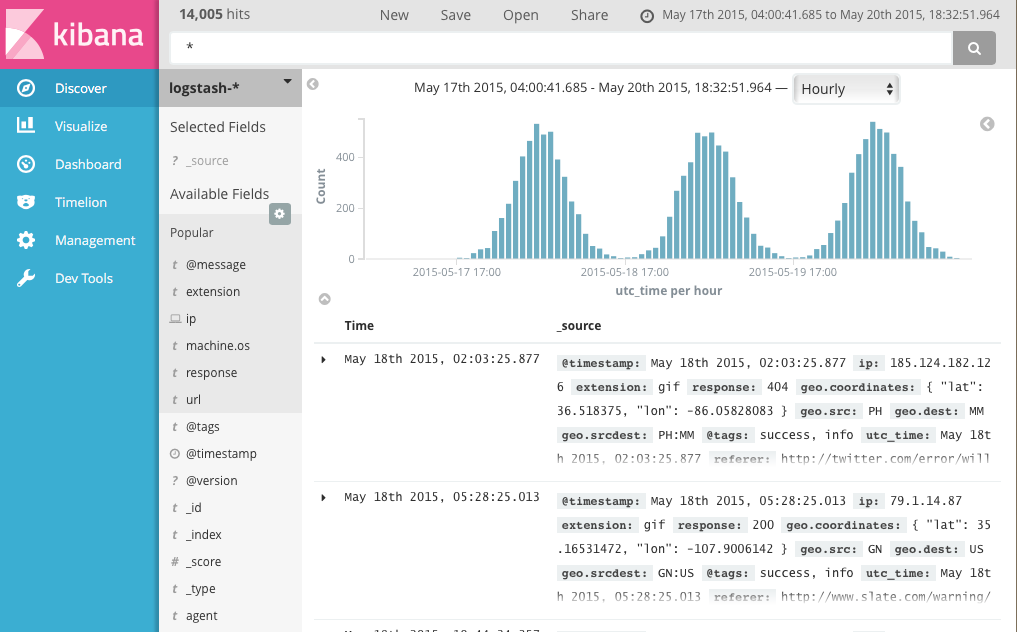


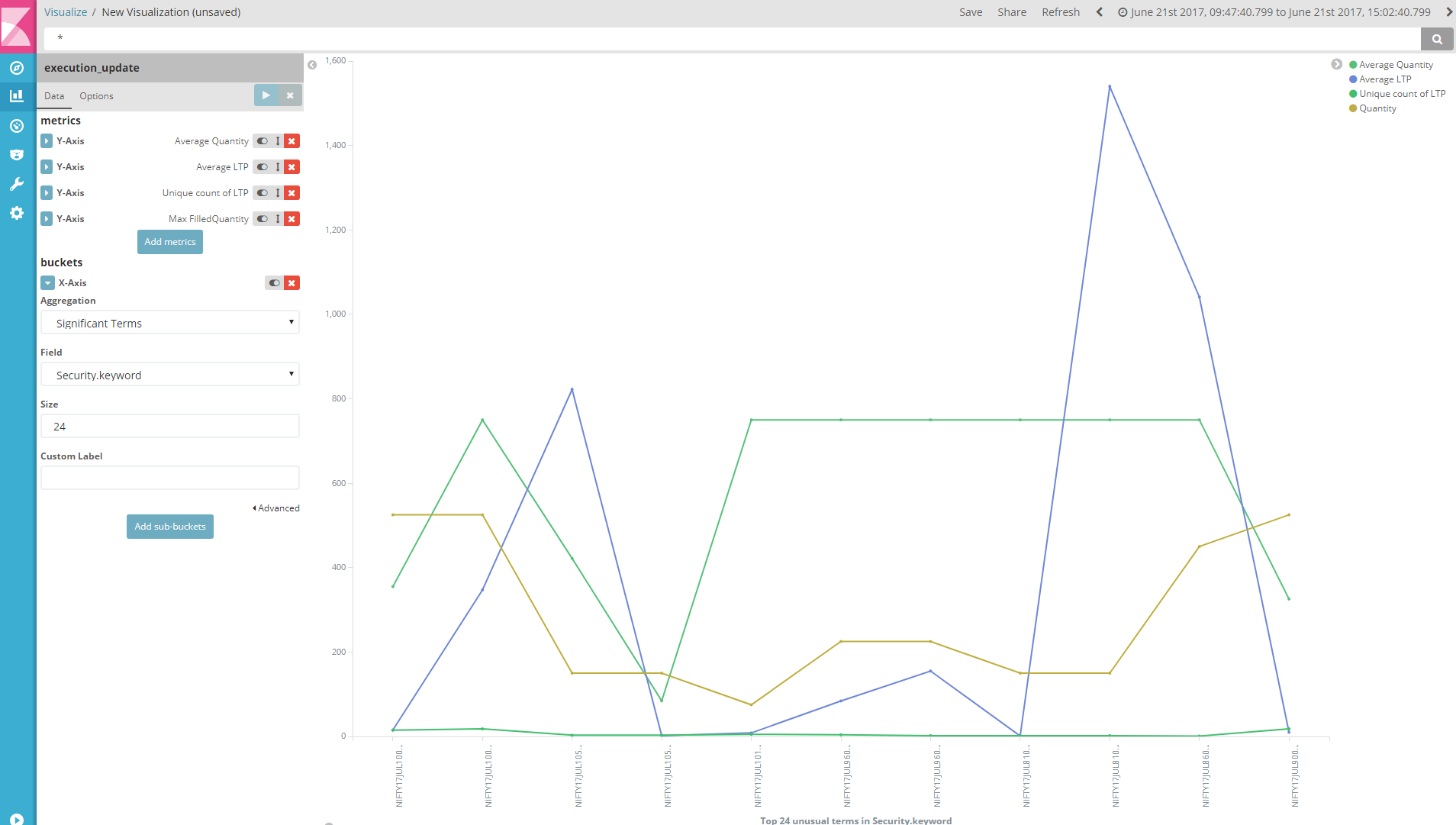










****