CHAPTER 1

INTRODUCTION

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 can be run on your own hardware or using our hosted Elasticsearch Service on Elastic Cloud, which is available on AWS and GCP. Elasticsearch is built using Java, and requires at least Java 8 in order to run. Only Oracle's Java and the OpenJDK are supported. The same JVM version should be used on all Elasticsearch nodes and clients.

Elasticsearch defaults to using /etc/elasticsearch for runtime configuration. The ownership of this directory and all files in this directory are set to root:elasticsearch on package installation and the directory has the setgid flag set so that any files and subdirectories created under /etc/elasticsearch are created with this ownership as well (e.g., if a keystore is created using the keystore tool).

Elasticsearch is a real-time distributed and open source full-text search and analytics engine. It is accessible from RESTful web service interface and uses schema less JSON (JavaScript Object Notation) documents to store data. It is built on Java programming language, which enables Elasticsearch to run on different platforms. It enables users to explore very large amount of data at very high speed.

The key concepts of Elasticsearch are as follows –

Node − It refers to a single running instance of Elasticsearch. Single physical and virtual server accommodates multiple nodes depending upon the capabilities of their physical resources like RAM, storage and processing power.

Cluster − It is a collection of one or more nodes. Cluster provides collective indexing and search capabilities across all the nodes for entire data.

Index − It is a collection of different type of documents and document properties. Index also uses the concept of shards to improve the performance. For example, a set of document contains data of a social networking application.

Type/Mapping − It is a collection of documents sharing a set of common fields present in the same index. For example, an Index contains data of a social networking application, and then there can be a specific type for user profile data, another type for messaging data and another for comments data.

Document − It is a collection of fields in a specific manner defined in JSON format. Every document belongs to a type and resides inside an index. Every document is associated with a unique identifier, called the UID.

Shard − Indexes are horizontally subdivided into shards. This means each shard contains all the properties of document, but contains less number of JSON objects than index. The horizontal separation makes shard an independent node, which can be store in any node. Primary shard is the original horizontal part of an index and then these primary shards are replicated into replica shards.

Replicas − Elasticsearch allows a user to create replicas of their indexes and shards. Replication not only helps in increasing the availability of data in case of failure, but also improves the performance of searching by carrying out a parallel search operation in these replicas.

Logstash – It is an open-source, centralized, events and logging manager. It is a part of the ELK (ElasticSearch, Logstash, Kibana) stack. Logstash can collect data from different sources and send to multiple destinations.

Logstash can also handle http requests and response data. Logstash provides a variety of filters, which helps the user to find more meaning in the data by parsing and transforming it. Logstash can also be used for handling sensors data in internet of things.Logstash is open source and available under the Apache license version 2.0

Kibana is the visualization layer of the ELK Stack — the world's most popular log analysis platform which is comprised of Elasticsearch, Logstash, and Kibana. Kibana Discover interface elements:

Search Bar: Use this to search specific fields and/or entire messages

Time Filter: Use this to filter logs based on various relative and absolute time ranges

Field Selector: Select fields to modify which ones are displayed in the *Log View*

Date Histogram: Bar graph under the search bar. By default, this shows the count of all logs, versus time (x-axis), matched by the search and time filter. You can click on bars, or click-and-drag, to narrow the time filter

Log View: Use this to look at individual log messages, and display log data filtered by fields. If no fields are selected, entire log messages are displayed

CHAPTER 2

LITERATURE SURVEY

Huai, Lee, Zhang, Xia, and Zhang [1]: The team addressed performance issues of big data analytics in distributed environments. They mainly argue that the current data analytics systems and applications are not efficient when analyzing distributed large data sets. So, they proposed the DOT framework, which extends the big data ecosystems in order to optimize handling large distributed data sets for analytics, concurrency control, and interaction between users and those data sets in real-time manner. Tests on DOT show its effectiveness, scalability, and fault-tolerance ability on complex analytic queries over MapReduce and Dryad.

Cheng, Qin, and Rusu [2]: The research work presented GLADE, which is a scalable distributed system for big data analytics. GLADE mainly takes analytical functions and executes them efficiently on the inputted data. The proposed system attempts to take full advantage of the parallelism available inside single machines as well as across a cluster of distributed computing nodes. The system demonstrations show that GLADE can outperform Hadoop in several querying and analytical operation scenarios, namely Average, k-means, Group by, and Top-K.

This is mainly because GLADE uses columnar storage and reads only the data needed for query execution while Hadoop normally reads all the data in the relation. In addition, GLADE employs point-to-point communication between the distributed nodes while Hadoop requires all-to-all communication among the nodes.

Mozafari, Zeng, D Antoni, and Zaniolo [3]: The authors was worked on extending XML's query language XPath. The study argues that in some scenarios, the well-known complex queries from distributed applications could be troublesome or impossible to be handled by XPath. Thus, the study proposed XSeq, which is an XML query language that extends the conventional XPath expressions and its dialects in order to optimize and ease querying large XML streams and exchanges.

Narang, etc.al [4]: The authors addressed the challenges of accuracy and speed in real-time co-clustering and collaborative filtering. They have proposed a hierarchical approach for distributed online and offline big data co-clustering and collaborative filtering. The approach has been tested online and offline on Netflix and Yahoo's KDD Cup datasets on a multi-core cluster infrastructure. The test results show that proposed approach, while maintaining high accuracy, outperforms all existing collaborative filtering mechanisms, both online and offline.

Chandramouli, Goldstein, and Duan [5]: The team proposed the TiMR framework. As temporal queries easy to specify and naturally real-time-ready, the introduced TiMR framework is based on the use of those queries. TiMR enables temporal queries to scale up to big offline datasets on existing MapReduce infrastructures. The main purpose of this research was to enhance the behavioral targeting (BT) mechanisms used in online-targeted advertisements and optimize it in real-time environments. The research experiments validate TiMR's high scalability and performance in the real-time BT applications domain.

Kumar, etc. al [6]: The recent success of several big data analytics-driven systems has created a massive interest in bringing such technological potential abilities to a wider variety of business domains. On the other hand, there are still big challenges in making these analytical systems easy to build and maintain. The team had introduced the open-source Hazy project, which targets these challenges through identifying the common patterns across domains that would ease and speed-up building the analytical systems and transferability among domains. The Haze code been employed by four enterprises as well as a research observatory at the South Pole.

Dhar. Etc.al [7]: The research work investigated the meaning of new terms as data science, importance of predictive modeling in big data environments, and how the Information Systems discipline can better support the needs of business managers in light of the emerging business intelligence, analytics technologies and ubiquitous big data infrastructures. The authors predict that data-savvy managers and professionals with deep analytical skills will be much needed for businesses but hard to find.

CHAPTER 3

SYSTEM DESIGN

Elasticsearch is mainly made up of clusters and nodes. Clusters are a collection of nodes that communicate with each other to read and write to an index. A cluster needs a unique name to prevent unnecessary nodes from joining. A node is a single instance of Elasticsearch. It usually runs one instance per machine. They communicate with each other via network calls to share the responsibility of reading and writing data. A master node organizes the entire cluster.Shards are individual instances of a Lucene index. Lucene is the underlying technology that Elasticsearch uses for extremely fast data retrieval.

3.1 PROPOSED SYSTEM:

ALGORITHMS:

i. Practical Scoring Function
ii. Inverted Index Algorithm

3.1.1 Practical Scoring Function

Elasticsearch uses the Boolean model to find matching documents, and a formula called the practical scoring function to calculate relevance. While this section mentions algorithms, formulae, and mathematical models, it is intended for consumption by mere humans. It uses indexing API's to traverse among millions of data in the documents. Elasticsearch runs Lucene under the hood so by default it uses Lucene's Practical Scoring Function. This is a similarity model based on Term Frequency (tf) and Inverse Document Frequency (idf) that also uses the Vector Space Model (vsm) for multi-term queries.

- score(q,d) is the relevance score of document d for query q.
- queryNorm(q) is the query normalization factor.
- coord(q,d) is the coordination factor.
- The sum of the weights for each term t in the query q for document d.

- tf(t in d) is the term frequency for term t in document d.
- idf(t) is the inverse document frequency for term t.
- t.getBoost() is the boost that has been applied to the query.
- norm(t,d) is the field-length norm, combined with the index-time field-level boost.

Now, let's get more familiar with each of the scoring mechanisms that make up the Practical Scoring Function:

3.1.1.1 Term Frequency

This is the square root of the number of times the term appears in the field of a document:

tf = sqrt(termFreq)

Term frequency clearly assumes that the more times a term appears in a document, the higher its relevancy should be. Usually that's the case and you'll probably continue to use this scoring mechanism, but if you just need to know that the term appears in the document at all and you don't care how many times, you can configure the field to ignore term frequency during indexing.

A better way to handle that situation, though, is to apply a filter using the term at query time. Note, too, that inverse document frequency can't be turned off so, even if you disable term frequency, the inverse document frequency will still play a role in the scoring. Finally, note that not-analyzed fields(typically those where you expect an exact match) will automatically have term frequency turned off.

3.1.2 Inverted Index Algorithm

3.1.2.1 Inverse Document Frequency

This is one plus the natural log (as in "logarithm", not "log file") of the documents in the index divided by the number of documents that contain the term:

idf = 1 + ln(maxDocs/(docFreq + 1))

What inverse document frequency captures is that, if many documents in the index have the term, then the term is actually less important than another term would be where few documents include the term.

3.1.2.2 Coordination

Counts the number of terms from the query that appear in the document.With the coordination mechanism, if we have a 3-term query and a document contains 2 of those terms, then it will be scored higher than a document that has only 1 of those terms. Like term frequency, coordination can be turned off, but that is typically only done when the terms are synonymous with each other (and therefore, having more than one of them does not increase relevancy). A better way to handle that situation, though, is to populate a synonym file to handle synonyms automatically.

3.1.2.3 Field Length Normalization

This is the inverse square root of the number of terms in the field:

```
norm = 1/sqrt(numFieldTerms)
```

For field length normailization, a term match found in a field with a low number of total terms is going to be more important than a match found in a field with a large number of terms. As with term frequency and coordination, you can choose to not implement field length norms in a document (the setting applies to all fields in the document). While you can save memory by turning this off, you may lose some valuable scoring input. The only time it might make sense to turn off this feature is similar to the case for turning off term frequency - when it does not matter how many terms there are, but only that the query term exists. Still, there are other ways of handling such a situation (like using a filter instead). Note that not-analyzed fields will have field length normalization disabled by default.

3.1.2.4 Query normalization

This is typically the sum of squared weights for the terms in the query.

Query normalization is used so that different queries can be compared. For any individual query, it uses the same score for every document (effectively negating its impact within an individual query) so it's not something we need to spend any time on.

3.1.2.5 Index Boost

This is a percentage or absolute number used to boost any field at index time. In practice an index boost is combined with the field length normalization so that only a single number will be stored for both in the index; however, Elasticsearch strongly recommends against using index-level boosts since there are many adverse effects associated with this mechanism.

3.1.2.6 Query Boost

This is a percentage or absolute number that can be used to boost any query clause at query time.Query boosting allows us to indicate that some part(s) of the query should be more important than other parts. Documents will be scored accordingly to their matches for each part. It can also be used to boost a particular index if you're searching across multiple indexes and want one to have more importance. There are quite a few options that can be used to boost a score at query time.

In practice, these boosts are combined with the queryNorm when applying explain, so the queryNorms of different values if you have used a boost at query time and performed an explain.The term frequency, inverse document frequency, and field-length normalization are stored for each document at index time. These are used to determine the weight of a term in a document.

3.2 ARCHITECTURE

The architecture of elastic stack comprises to the following components:

- Clusters
- Nodes
- Shards
- Replicas

The clusters are the main storage system of the elasticsearh database. A cluster can hold a single node or even multiple nodes. Nodes can be created from different applications or databases by using several logstash instances. Shards are units present inside the nodes which hold the data. Replicas are also known as backup shards which can be used to avoid inconsistency and data loss. Pipelines are created to retrieve data from various sources of data and export them to Elasticsearch.

The architecture of the elastic represented in figure 3.2.

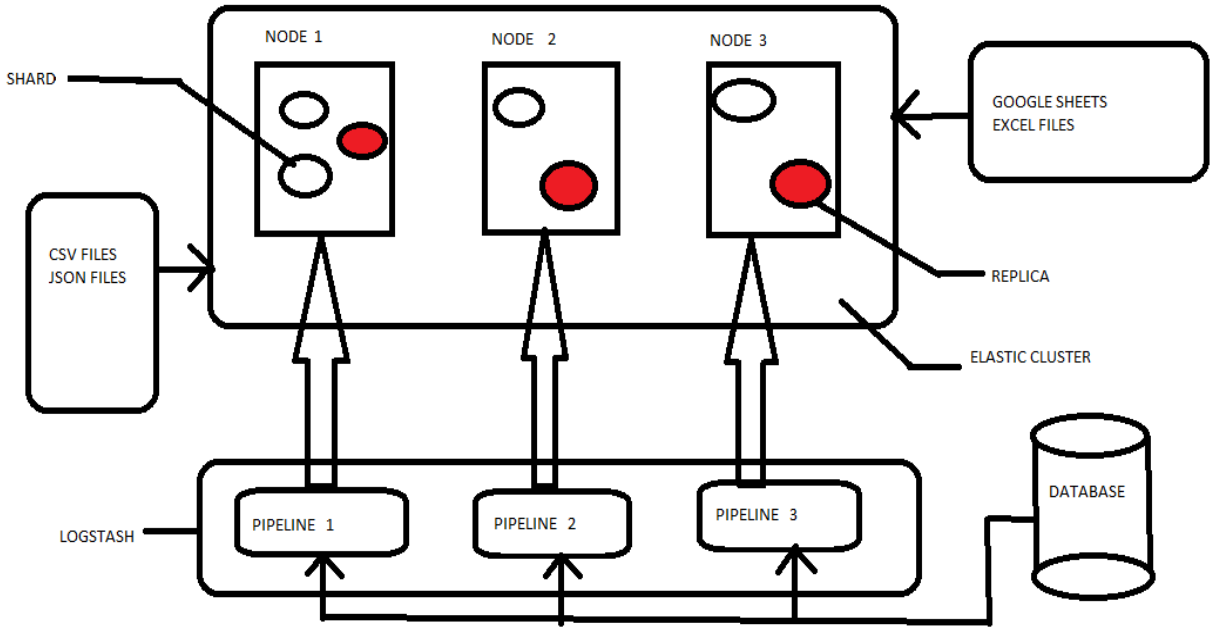

Figure 3.2 Elastic stack Architecture

3.3 Working Principle

3.3.1 How Elasticsearch Routing Works

Understanding routing is important in large elasticsearch clusters. By exercising fine-grained control over routing the quantity of cluster resources used can be severely reduced, often by orders of magnitude.

The primary mechanism through which elasticsearch scales is sharding. Sharding is a common technique for splitting data and computation across multiple servers, where a property of a document has a function returning a consistent value applied to it in order to determine which server it will be stored on. The value used for this in elasticsearch is the document's _id field by default.
The algorithm used to convert a value to a shard id is what's known as a consistent hashing algorithm.

Maintaining good cluster performance is contingent upon even shard balancing. If data is unevenly distributed across a cluster some machines will be over-utilized while others will remain mostly idle. To avoid this, we want as even a distribution of numbers coming out of our consistent hashing algorithm as possible. Document ids hash well generally because they are evenly distributed if they are either UUIDs or monotonically increasing ids (1,2,3,4 …).

This is the default approach, and it generally works well as it solves the problem of evening out data across the cluster. It also means that fetches for a single document only need to be routed to the shard that document hashes to. But what about routing queries? If, for instance, we are storing user history in elasticsearch, and are using UUIDs for each piece of user history data, user data will be stored evenly across the cluster. There's some waste here, however, in that this means that our searches for that user's data have poor data locality.

Queries must be run on all the shards within the index, and run against all possible data. Assuming that we have many users we can likely improve query performance by consistently routing all of a given user's data to a single shard. Once the user's data has been so-segmented, we'll only need to execute across a single shard when performing operations on that user's data.

3.3.2 Implementing Custom Routing

Specifying routing information in elasticsearch is fairly straightforward. There are two strategies that may be employed:

- Routing based on a certain document field's value.
- Routing based on a value passed at index time via the routing query parameter.

Queries must always include the routing query parameter to take advantage of explicit routing. We'll start with the first approach, as it is simpler.

An important best practice is to ensure that elasticsearch's default routing is disabled for our index. Once custom routing is in play it should be used exclusively in order to preserve developer sanity. Disabling the default ID based routing can be achieved through the mapping API and will be enforced in subsequent examples.

A micro-blogging platform is used as an example in this exercise. We call this fictional microblogging platform microblagh. The case that will be optimized for will be searches scoped to a single user. By routing based on the user_id value we'll be able to make searches limited to a single user's posts much faster. To start, we must define an index and a mapping type post, with _routing.required set to true. We also specify _routing.path, which tells elasticsearch which field value to look at for routing. With this setting enabled, attempts to index documents without explicitly specified routing will fail with an error.

3.3.3 Schema with Routing Enforcement

Creating documents in our schema is straightforward enough, and nothing special need be done.In order to test our routed index, let's populate it with a few sample documents.

```
POST /microblagh
{"mappings":
{"post":
{"_routing": {"required": true, "path": "user_id"},
 "properties": {
 "user_id": {"type": "integer"},
 "username": {"type": "string"},
 "text": {"type": "string"}}}}}
```

3.3.4 Sample Data for Microblagh

```
POST /_bulk
{"index": {"_index": "microblagh", "_type": "post", "_id": 1}
{"user_id": 1, "username": "albert", "text": "all work, and no play"}
{"index": {"_index": "microblagh", "_type": "post", "_id": 2}
{"user_id": 1, "username": "albert", "text": "make jack a dull boy"}
{"index": {"_index": "microblagh", "_type": "post", "_id": 3}
{"user_id": 2, "username": "john", "text": "Microblagh 4eva"}
{"index": {"_index": "microblagh", "_type": "post", "_id": 4}
{"user_id": 1, "username": "stacy", "text": "Microblagh is overrated"}
```

We can test that documents have been properly routed by running operations with the routingquery parameter set. For instance, post id 3 should be routed based on its user_id value of 2. To retrieve it, we would run GET /microblagh/post/3. To retrieve it, explicitly routing the request *only* to the shards it actually exists on, GET /microblagh/post/3?routing=2 can be run. Finally, we can verify the properties of routing by running GET /microblagh/post/3?routing=1, which should return a 404error, as we have incorrectly specified the routing value.

The same effects can be seen with queries as well. Searching for the text Microblagh, across the entire index should return documents 3 and 4, by stacy and john. Since they have different routing values, however, only one is returned if routing is set.

```
POST /microblagh/post/_search
{"query": {"match": {"text": "Microblagh"}}}
POST /microblagh/post/_search?routing=2
{"query": {"match": {"text": "Microblagh"}}}
POST /microblagh/post/_search?routing=2,1
{"query": {"match": {"text": "Microblagh"}}}
```

3.3.5 Performance Concerns

It should be noted that to write a search for a single user, we would still need to have a filter on our query scoping it to that user specifically. Routing should not be used to limit logical results of data, since changing the number of underlying shards will change which data is visible for a given routing value.

The cardinality of routing values should generally be a large multiple of the number of shards to enable balanced data. This ensures that both distribution of data and computation is even across the cluster.

Regarding the choice of using field values for routing, consideration should be given to explicitly specifying routing values at index time via the routing query parameter rather than relying on field values. A small speed increase can be achieved when using the query parameter rather than as a field value. At scale this difference could be non-negligible.

3.3.6 Clustering and Index Internals

It's almost impossible to truly understand elasticsearch's clustering features without also understanding the internals of Lucene indexes. The elastic part of elasticsearch's name is in reference to its clustering capabilities. It's capacity to seamlessly add and remove servers and expand capacity. While Lucene does have some provisions for operations on multiple servers, they are quite basic, and not nearly as scalable.

Elasticsearch's clustering features can be neatly divided into two categories, durability and scalability. Before we explore these topics however, let's get dive into the internals of a Lucene index.

3.3.7 Index Internals

Indexes are the central piece of data to which computation is applied in elasticsearch. Everything involving data in elasticsearch occurs at the index level. While complex, there are a few things about the internals of elasticsearch indexes that are quite useful to know. A cursory knowledge of the implementation and architecture of elasticsearch

indices becomes important when considering clustering, capacity planning, and performance optimization.

Let's start by discussing Lucene indexes, upon which elasticsearch indexes are built. A Lucene index is subdivided into a variable number of segments at any given time. Each of these segments is a completely separate index in and of itself.

Lucene indexes create more segments as documents are added, and when they become more numerous tries to merge them back into fewer segments. The smaller the number of segments, the faster operations run (one segment is optimal). Merging, however, has costs as well, so Lucene attempts to merge segments at a rate where merge costs and search efficiency are balanced.

It is due to this architecture that searches be seamlessly executed over multiple indexes. A search over 2 indexes with 1 segment apiece is almost identical to a search over 1 index with 2 segments.

Elasticsearch takes Lucene index/segment symmetry one step farther, leveraging Lucene's ability to span operations over indexes to implement its clustering support. When we speak of an index in elasticsearch, we are usually talking about elasticsearch's index abstraction which sits atop multiple Lucene indexes. Each elasticsearch index is divided into a number of shards (5 by default). Each of these shards contains a unique portion of the documents in the elasticsearch index.

A shard itself is a single logical index, but is comprised of a number of Lucene indexes–a primary and a configurable number of replicas–all of which contain the same documents, but are full Lucene indexes in and of themselves. Multiples are created to allow for both durability guarantees and distributed search scalability across clusters.

When a document is added to the elasticsearch index, it is routed to the proper shard based on its id. When a search is executed it is run in parallel over all the shards in an index (on either a primary, or replica Lucene index), and then the results are combined. This means

that splitting your documents over one elasticsearch index with 5 shards is equivalent to manually splitting your data over 5 elasticsearch indexes with one shard.

Analysis in elasticsearch has no special magic, it mostly relies on the same principles underlying efficient data storage and retrieval in traditional relational systems. That is to say, it is contingent on efficient traversal of sorted trees.

3.3.8 Index Durability

Durability in elasticsearch is implemented by its replica feature, whereby data is mirrored to multiple servers simultaneously. By default elasticsearch indexes have a replica count value set to one. This means that each piece of data will exist on at least 2 servers in a running elasticsearch cluster, once on a primary, and once on a secondary location. Upping the replica count to 4 would mean that same piece of data would be guaranteed to exist on at least 5 separate servers.

Should a server fail, elasticsearch will self-heal. Given a replica count of 1, and a cluster consisting of 3 servers, it will be the case that each server will have 2/3 of the cluster data available. In this scenario the loss of a single server will be tolerated without data-loss.

If a single server in that setup were to fail the cluster state, visible at the cluster health endpoint */cluster/_health* would change from green to yellow. Some data on the cluster would only be present on a single server, which would cause elasticsearch to attempt to re-balance the replicas, dividing replica indexes evenly between the 2 remaining servers. Should the third server be fixed and added back in, elasticsearch would re-migrate the data back across all 3 servers. If two of the three servers were to fail, the state would change from yellow to red.

3.3.9 Write Durability

Write consistency in elasticsearch may be specified per write-request, or on a per-node basis with the action.write_consistency setting. In an elasticsearch cluster one may write data at one of three consistency levels: all, quorum, and one, with decreasing guarantees for data-durability.

The one consistency level is easy enough to understand: a single node will receive the data and persist it before acknowledging the write. After that point, the data will eventually be replicated to all replicas of the shard. The all consistency is similarly simple; each and every replica in the shard acknowledges the write before a response is returned. Lastly, when using the default quorum consistency level a majority of shards within the cluster must acknowledge the write before a response is returned. The various stages of a write, and the points at which each consistency is achieved is illustrated in the figure below.

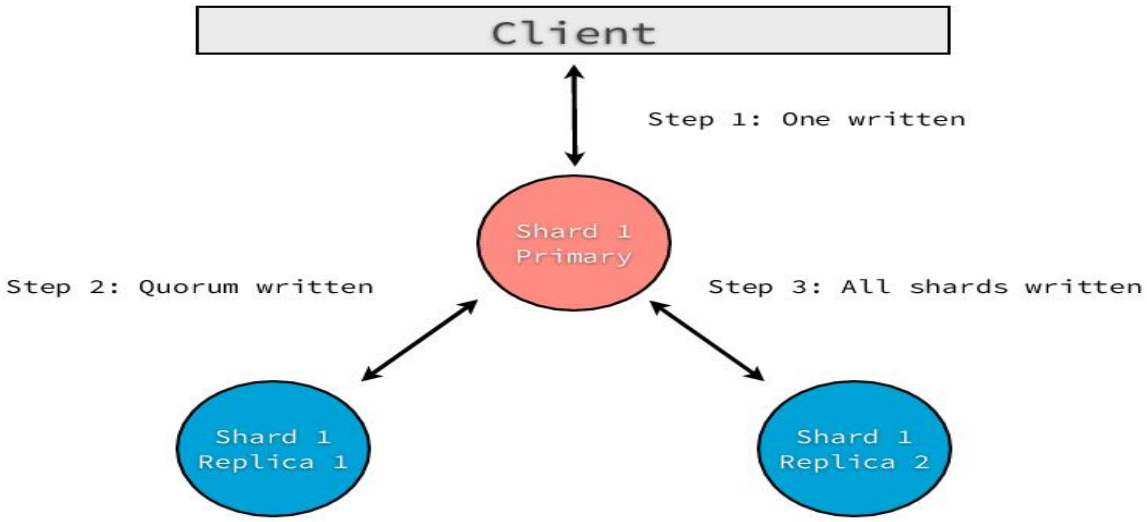

Figure 3.3.9 Write Durability

Each consistency level has its uses. The one consistency level will return most quickly, followed by the slower quorum consistency level, and finally the all consistency level, which is slowest of all. For most applications the quorum consistency level is a good trade-off between safety and performance.

Consistency levels can be specified during write operations.

3.3.10 Updating A Document With An Explicit Consistency Level

PUT /atest/atype/adocument?consistency=one

{"afield": "avalue"}

If occasional losses of data during unusual scenarios–such as cluster maintenance and the occasional outage–are okay, then a consistency level of one may be sufficient. If a node goes down soon after being written to there is a chance that it may not have had time to replicate its most recent writes. In this case data loss may occur. The quorum consistency level is quite safe given a sufficient replica count.

With a replica count of two, for instance, there is a total of three shards. Data will then have to be present on both the primary shard and at least one of the two replica shards before returning. When running with either a low replica count or in situations where data integrity is paramount the all consistency setting may be appropriate. If the writes are slow consider either lowering the consistency level or increasing the number of machines/shards in your cluster.

If the primary data-store is not elasticsearch it is almost certain that the all write-level is overkill.The performance impact will likely not be worth the extra guarantees of durability. As a final point, it should be noted that during a split-brain failure, where a single cluster has divided into two due to either a network partition or overloading, write-consistency guarantees will be rendered moot for obvious reasons.

CHAPTER 4
IMPLEMENTATION DETAILS AND RESULTS

The system is proposed to have the following modules. They are

- Setting up Elasticsearch.
- Ingesting data using Logstash.
- Visualizing using Kibana.

4.1 Setting up Elasticsearch

Elasticsearch is a nosql database that can be downloaded and installed from elastic.co. Elasticsearch supports all types of operating system.The important part of Elasticsearch is setting up the cluster. An Elasticsearch cluster is a group of nodes that have the same cluster.name attribute. As nodes join or leave a cluster, the cluster automatically reorganizes itself to evenly distribute the data across the available nodes.

If a single instance of Elasticsearch is being run, the cluster must comprise of one node. All primary shards reside on the single node. No replica shards can be allocated, therefore the cluster state remains yellow. The cluster is fully functional but is at risk of data loss in the event of a failure. Add nodes to a cluster to increase its capacity and reliability. By default, a node is both a data node and eligible to be elected as the master node that controls the cluster. A new node can be configured for a specific purpose, such as ingest requests.

When more nodes are added to a cluster, it automatically allocates replica shards. When all primary and replica shards are active, the cluster state changes to green. To add a node to a cluster it is necessary to:

Step 1: Set up a new Elasticsearch instance.

Step 2: Specify the name of the cluster in its cluster.name attribute. For example, to add a node to the logging-prod cluster, set cluster.name: "logging-prod" in elasticsearch.yml.

Step 3: Start Elasticsearch. The node automatically discovers and joins the specified cluster. The document API's can be used in the elasticsearch index.

Single Document APIs are

- Index API
- Get API
- Delete API
- Update API

Multi Document APIs are

- Multi Get API
- Bulk API
- Delete By Query API
- Update By Query API
- Reindex API

Elasticsearch is provided with different plugins. It uses standard RESTful APIs and JSON. We also build and maintain clients in many languages such as Java, Python, .NET, SQL, and PHP.

4.2 Ingesting Data Using Logstash

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.

4.2.1 Inputs

Inputs are used to get data into Logstash. Some of the more commonly-used inputs are,

(1) File: It reads from a file on the filesystem, much like the UNIX command tail -0F

(2) Syslog: It listens on the well-known port 514 for syslog messages and parses according to the RFC3164 format

(3) Redis: It reads from a redis server, using both redis channels and redis lists. Redis is often used as a "broker" in a centralized Logstash installation, which queues Logstash events from remote Logstash "shippers".

(4) Beats: It processes events sent by Beats.

4.2.2 Filters

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:

(1) Grok

It 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.

(2) Mutate

It perform general transformations on event fields. You can rename, remove, replace, and modify fields in your events.

(3) Drop: drop an event completely, for example, *debug* events.

(4) Clone: make a copy of an event, possibly adding or removing fields.

(5) Geoip: add information about geographical location of IP addresses.

4.2.3 Output

(1) Elasticsearch is the most commonly used output filter.

4.3 Visualizing Using kibana

Kibana is the visualization layer of the ELK Stack — the world's most popular log analysis platform which is comprised of Elasticsearch, Logstash, and Kibana.

Kibana querying is an art unto itself, and there are various methods for performing searches on your data. This step will describe some of the most common search methods as well as some tips and best practices that should be memorized for optimized user experience.

4.3.1 Free-Text Search

Free text search works within all fields including the source field, which includes all the other fields. If no specific field is indicated in the search, the search will be done on all of the fields that are being analyzed.

In the search field at the top of the Discover page, run these searches and examine the result set the time parameter on the top right of the dashboard to the past month to capture more data

- category
- Category
- categ
- cat*
- category
- “category”
- category/health
- “category/health”
- Chrome

Text searches are not case sensitive. This means that "category" and "CaTeGory" will return the same results. When you put the text within double quotes (""), you are looking for an exact match, which means that the exact string must match what is inside the double quotes. This is why [category\/health] and ["category/health"] will return different results

Kibana wildcard searches—you can use the wildcard symbols [*] or [?] in searches. [*] means any number of characters, and [?] means only one character

4.3.2 Field-Level Search

Another common search in Kibana is field-level queries, used for searching for data inside specific fields. To use this type of search that, you need to use the following format:

<fieldname>:search

As before, run the following searches to see what you get (some will purposely return no results):

- name:chrome
- name:Chrome
- name:Chr*
- response:200
- bytes:65
- bytes:[65 TO *]
- bytes:[65 TO 99]
- bytes:{65 TO 99}
- _exists_:name

1. Field-level searches depend on the type of field.
2. A range can be searched within a field. If [] is used the results are inclusive. If {} is used the results are exclusive.
3. Using the _exists_ prefix for a field will search the documents to see if the field exists.
4. When using a range, a very strict format must be followed and capital letters must be usedto specify the range.

4.3.3 Logical Statements

Logical statements can be used in searches in these ways:

- USA AND Firefox
- USA OR Firefox
- (USA AND Firefox) OR Windows
- -USA

- !USA
- +USA
- NOT USA

1. Proper format such as capital letters must be used to define logical terms like AND or OR.
2. Parentheses can be used to define complex and logical statements.
3. – , ! and NOT can be used to define negative terms.

4.3.4 Kibana Special Characters

All special characters need to be properly escaped. The following is a list of all available special characters,

+ - && || ! ( ) { } [ ] ^ " ~ * ? : \

4.3.5 Proximity searches

Proximity searches are an advanced feature of Kibana that takes advantage of the Lucene query language. [categovi~2] means a search for all the terms that are within two changes from [categovi]. This means that all category will be matched. Proximity searches use a lot of system resources and often trigger internal circuit breakers in Elasticsearch. If you try something such as [catefujt~10], it is likely not to return any results due to the amount of memory used to perform this specific search.

To assist users in searches, recent versions include a filtering dialog that allows easier forming of Kibana search syntax.

- To use the dialog, simply click the Add a filter + button under the search box and begin experimenting with the conditionals. The Console editor allows writing multiple requests below each other. As shown in the Console section, you can submit a request to Elasticsearch by positioning the cursor and using the Action Menu. Similarly you can select multiple requests in one go:

- Console will send the request one by one to Elasticsearch and show the output on the right pane as Elasticsearch responds. This is very handy when debugging an issue or trying query combinations in multiple scenarios.
- Selecting multiple requests also allows you to auto format and copy them as cURL in one go.

4.4 System Specification

The system specification contained the requirements of the application. The requirements divided further divided into the software and hardware.

4.4.1 Hardware Requirements

PROCESSOR : Intel Core i7

RAM : 16 GB RAM

DISK SPACE : 1 TB

4.4.2 Software Requirements

The technologies selected for implementing this project are

- Kibana
- Logtsash
- Beats
- Elasticsearch

4.5 Results

Figure 4.1 Proximity searches

Application programming interfaces can be created by using various programming languages.The output dashboard can be used as iframes in hyper text mark up languages.

4.5.1 Index Management and Index Patterns

An index pattern can match the name of a single index, or include a wildcard (*) to match multiple indices.

For example, Logstash typically creates a series of indices in the format logstash-YYYY.MMM.DD. To explore all of the log data from May 2018, you specify the index pattern logstash-2018.05*.

Create patterns for the vtiger data set, which has an index named shakespeare, and the accounts data set, which has an index named bank. These data sets don't contain time-series data.

1. In Kibana, open Management, and then click Index Patterns.
2. If this is your first index pattern, the Create index pattern page opens automatically. Otherwise, click Create index pattern in the upper left.
3. Enter vtiger* in the Index pattern field.

1. Click Next step.
2. In Configure settings, click Create index pattern. For this pattern, you don't need to configure any settings.
3. Define a second index pattern named ba* You don't need to configure any settings for this pattern.

Now create an index pattern for the Logstash data set. This data set contains time-series data.

1. Define an index pattern named logstash*.
2. Click Next step.
3. In Configure settings, select @timestamp in the Time Filter field name dropdown menu.
4. Click Create index pattern.

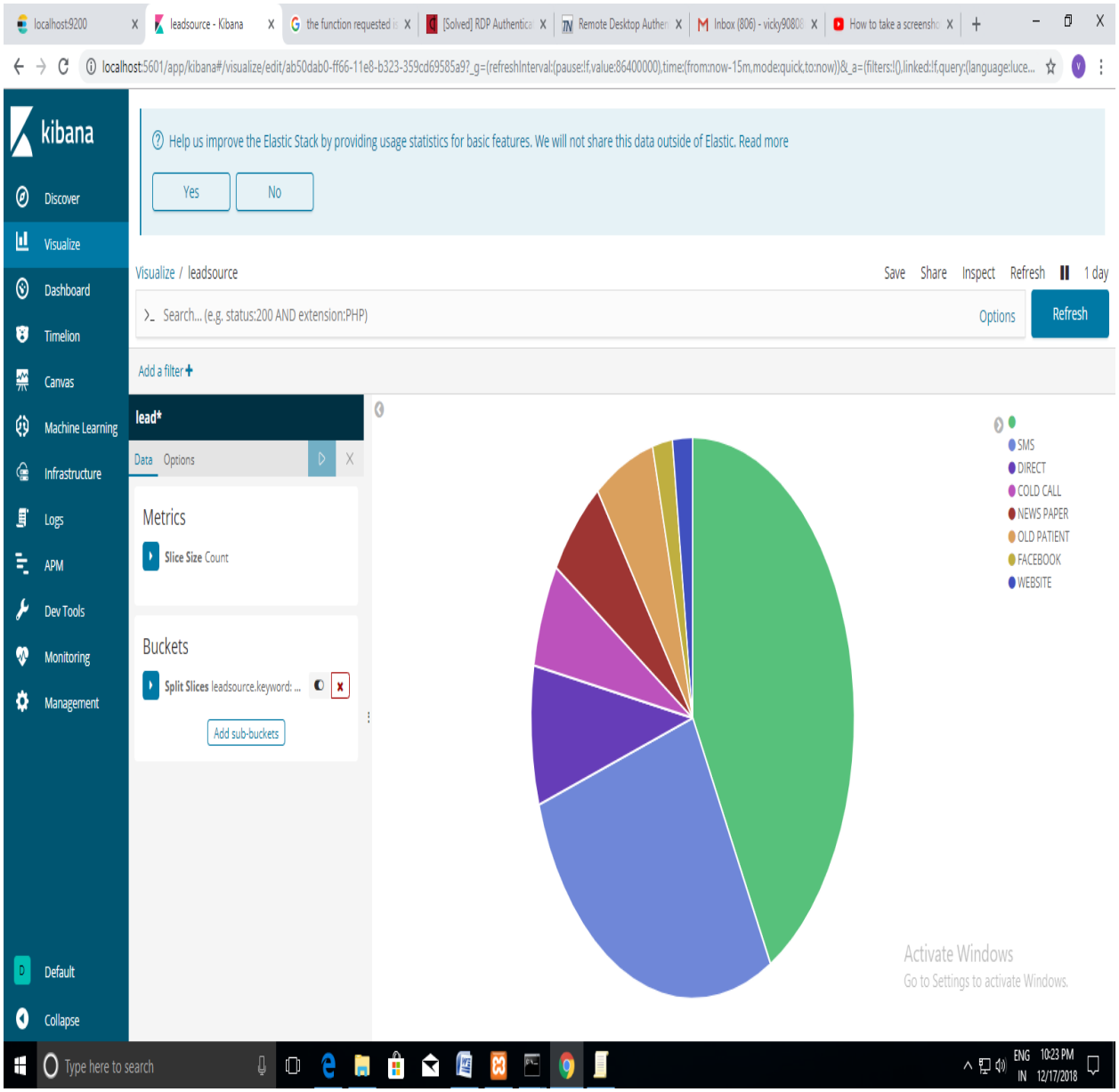

Figure 4.2 Index Management

4.5.2 Creating Visualizations

1. Open **Visualize.**
2. Click **Create a visualization** or the + button. You'll see all the visualization types in Kibana.

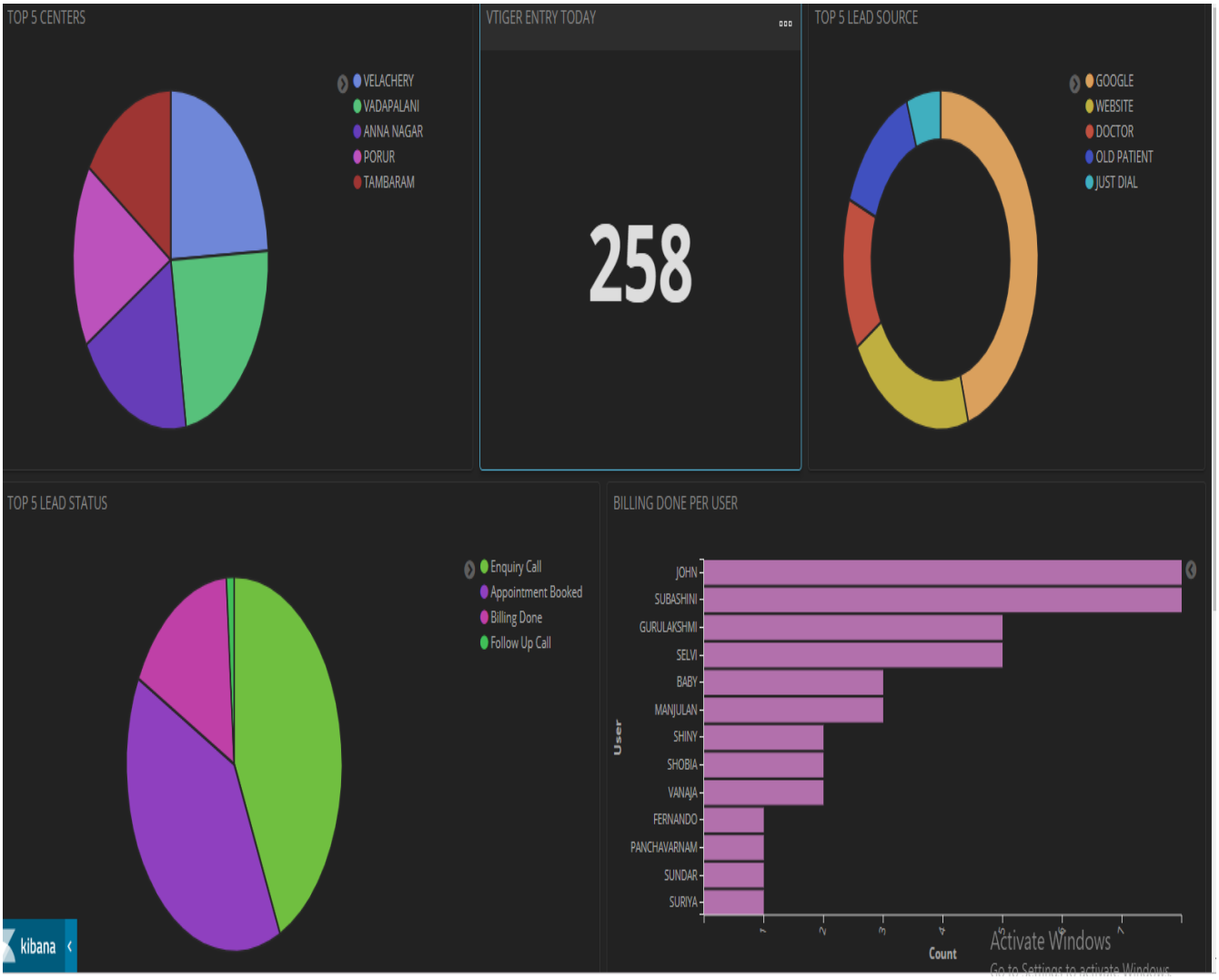

Figure 4.3 Combining Visualization

3. Click Pie or any deired visualization.

4. In **New Search**, select any index pattern. The pie chart can be used to gain insight into the account balances in the bank account data.

4.5.3 Creating Dashboards

1. Add the visualizations to a single dashboard and save the dashboard.

2. The dashboards and visualizations can be altered using the timestamp filter.

3. The dashboardscan be exported as pdf, iframes and images

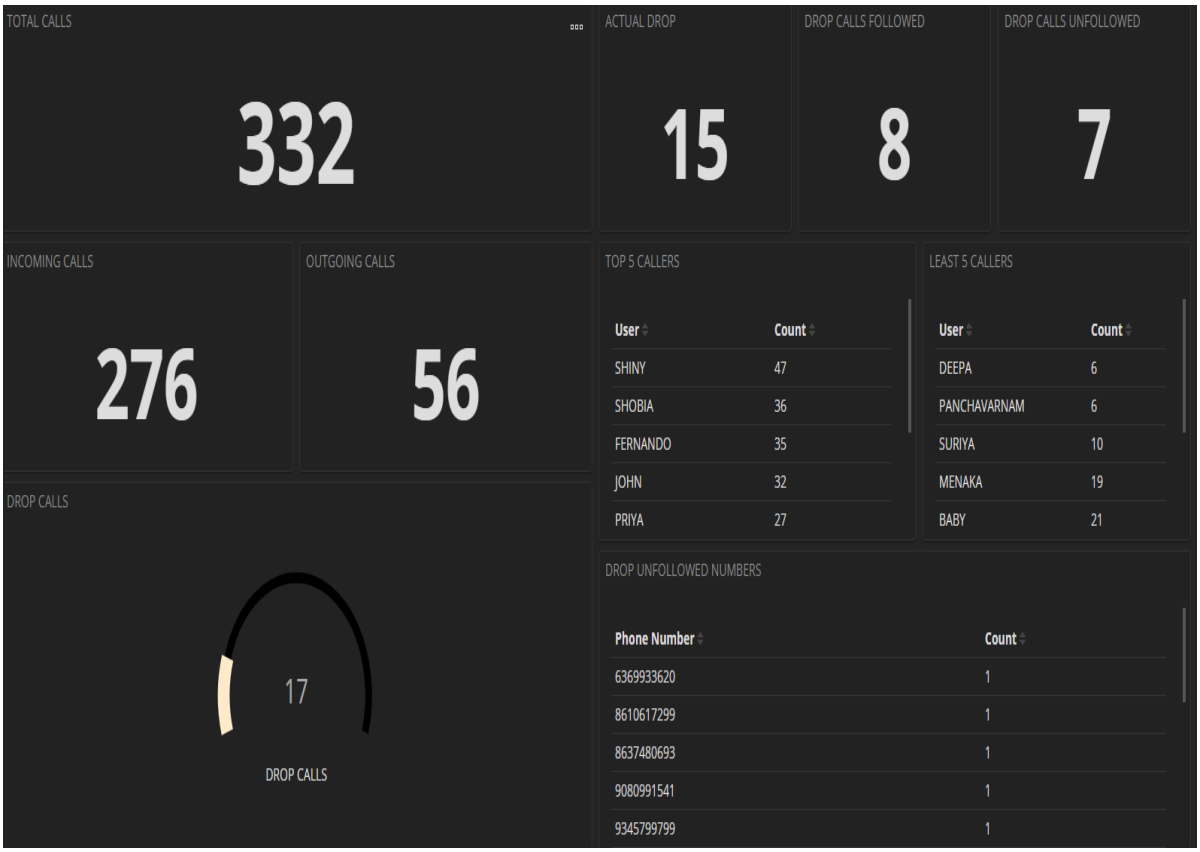

Figure 4.4 Creating Dashboards

3. The refresh interval can be altered to get the current stats and work progress of applications.

4. The dashboards can be combined together to create business intelligence.

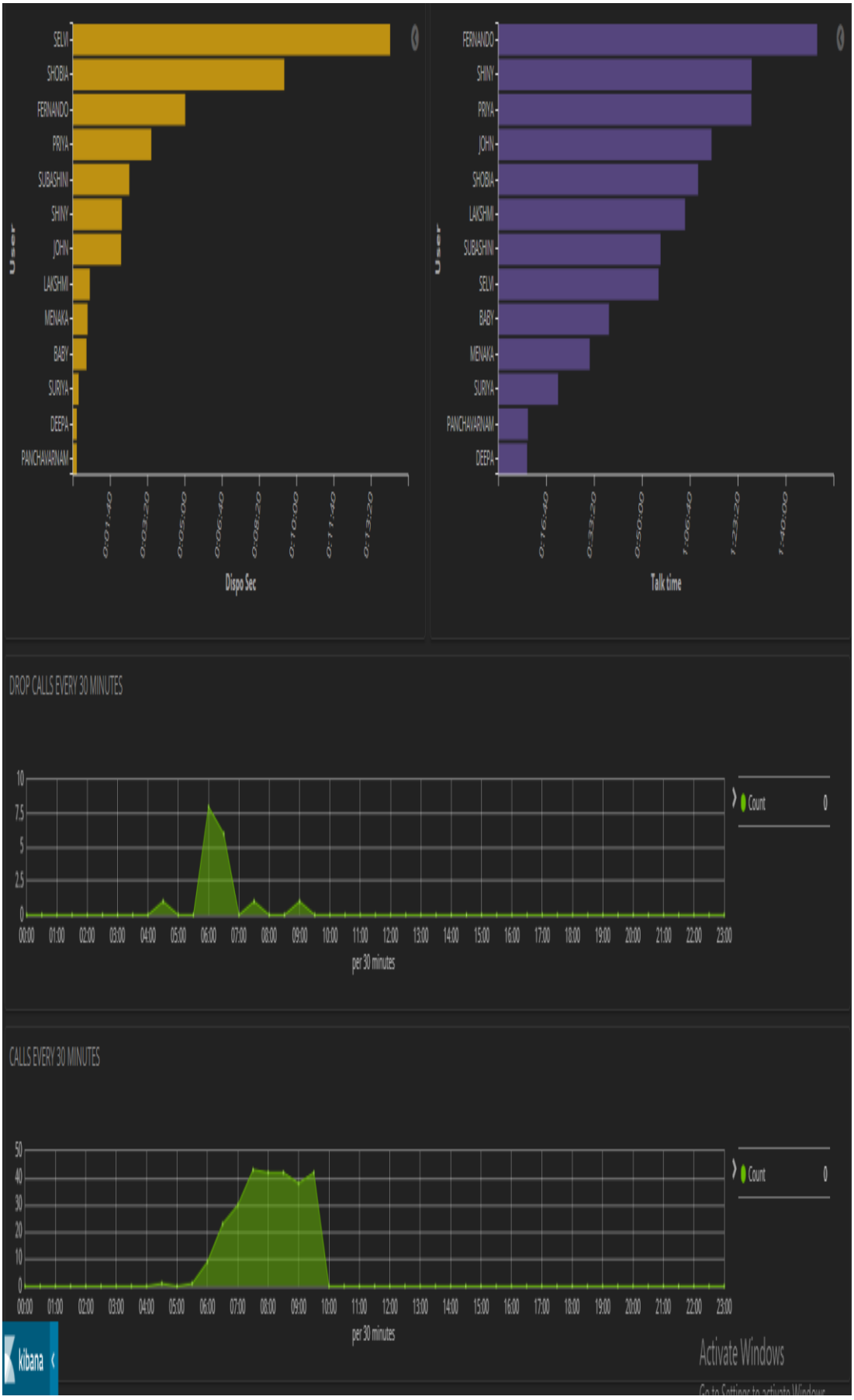

Figure 4.5 Timelion Series

CHAPTER 5

CONCLUSION

The Data analytics using Elastic Stack provides real time statistical reports which will be highly beneficial for organizations. Most of the search engines are created using the Elasticsearch database. The main scope of the project is to retrieve millions of data from various applications dynamically and perform analysis according to the requirement of the end user.By using this project the end user can gather relevant information from all his applications and make changes at the earliest.Any error in an application can be detected and rectified easily.

REFERENCES

[1] Dhar, V. (2013). Data Science and Prediction. Communications of the ACM , 56 (12), 64-73. Elgendy, N., & Elragal, A. (2014).

[2] Big Data Analytics: A Literature Review Paper. The 14th Industrial Conference on Data Mining (ICDM). Petersburg: Springer-LNCS. Fisher, D., DeLine, R., Czerwinski, M., & S., D. (2012, May-June). Interactions With Big Data Analytics. Interactions , 50-59.

[3] Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., et al. (2013). BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. SIGMOD (pp. 1197-1208).

[4] Efficient Skyline Computation on Big Data. TKDE , 25 (11), 2521-2535. He, Y., Lee, R., Huai, Y., Shao, Z., Jain, N., Zhang, X., et al. (2011).

[5] RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. The 27th International Conference on Data Engineering (ICDE) (pp. 1199-1208). IEEE.

[6] Herodotou, H., Lim, H., Luo, G., Boisov, N., Dong, L., Cetin, F., et al. (2011). Starfish: A Self-tuning System for Big Data Analytics. 5th Biennial Conference on Innovative Data Systems Research (CIDR '11), (pp. 261-272). CA.

[7] Huai, Y., Lee, R., Zhang, S., Xia, C., & Zhang, X. (2011). DOT: A Matrix Model for Analyzing