Elasticsearch is the world's most advanced search and analytics engine. It has the ability to make massive amounts of data usable in a matter of milliseconds. It not only gives you the power to build blazingly fast search solutions over a massive amount of data, but can also serve as a NoSQL data store.

This guide will assist you to quickly become a competent developer with a solid knowledge base and an understanding of the Elasticsearch core concepts. At the beginning, this book will cover the fundamental concepts required to start working with Elasticsearch, and then it will take you through more advanced concepts of search techniques and data analytics. This book provides complete coverage of working with Elasticsearch using Python and Java APIs to perform CRUD operations, aggregation-based analytics, handling document relationships, working with geospatial data, and controlling search relevancy. In the end, you will not only learn about scaling Elasticsearch clusters in production but also how to secure Elasticsearch clusters and take data backups using best practices.

Who this book is written for
Anyone who wants to build efficient search and analytics applications can choose this book. It is also beneficial for skilled developers, especially ones experienced with Lucene or Solr, who now want to learn Elasticsearch quickly.

What you will learn from this book
- Understand advanced Elasticsearch concepts and REST APIs
- Write CRUD operations and other search functionalities using the ElasticSearch Python and Java clients
- Dig into a wide range of queries and find out how to use them correctly
- Design schema and mappings with built-in and custom analyzers
- Excel in data modeling concepts and query optimization
- Master document relationships and geospatial data
- Build analytics using aggregations
- Set up and scale Elasticsearch clusters using best practices
- Learn to take data backups and secure Elasticsearch clusters

Harness the power of Elasticsearch to build and manage scalable search and analytics solutions with this fast-paced guide
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Aggregations for Analytics'
- A synopsis of the book’s content
- More information on Elasticsearch Essentials
About the Author

Bharvi Dixit is an IT professional with an extensive experience of working on the search servers (especially Elasticsearch) and NoSQL databases. He is currently working as a technology and search expert with GrownOut, a SAAS-based referral hiring solution provider company. He is the organizer and speaker of Delhi's Elasticsearch Meetup Group, which is one of the fastest growing Elasticsearch communities in India.

He also works as a freelance Elasticsearch consultant and has helped many small to medium size organizations in adapting Elasticsearch for different use cases such as, creating search solutions for big data-automated intelligence platforms in the area of counter-terrorism and risk management as well as in other domains such as recruitment, e-commerce, finance and log monitoring.

He holds a master's degree in computer science from LBSIM - Delhi, India, and has a keen interest in creating scalable backend platforms. His other interest area are data analytics, distributed computing, automations, and DevOps. Java and Python are the primary languages in which he loves to write code, and he has already built a proprietary software for consultancy firms.

In his spare time, he loves writing blogs and reading the latest technology books. He can be connected through LinkedIn at https://in.linkedin.com/in/bharvidixit.
Preface

With constantly evolving and growing datasets, organizations have the need to find actionable insights for their business. Elasticsearch, which is the world's most advanced search and analytics engine, brings the ability to make massive amounts of data usable in a matter of milliseconds. It not only gives you the power to build blazingly fast search solutions over a massive amount of data, but can also serve as a NoSQL data store.

Elasticsearch Essentials will guide you to become a competent developer quickly with a solid knowledge and understanding of the Elasticsearch core concepts. In the beginning, this book will cover the fundamental concepts required to start working with Elasticsearch and then it will take you through more advanced concepts of search techniques and data analytics.

This book provides complete coverage of working with Elasticsearch using Python and Java APIs to perform CRUD operations, aggregation-based analytics, handling document relationships, working with geospatial data, and controlling search relevancy.

In the end, you will not only learn about scaling Elasticsearch clusters in production, but also how to secure Elasticsearch clusters and take data backups using best practices.
What this book covers

Chapter 1, Getting Started with Elasticsearch, provides an introduction to Elasticsearch and how it works. After going through the basic concepts and terminologies, you will learn how to install and configure Elasticsearch and perform basic operations with Elasticsearch.

Chapter 2, Understanding Document Analysis and Creating Mappings, covers the details of the built-in analyzers, tokenizers, and filters provided by Lucene. It also covers how to create custom analyzers and mapping with different data types.

Chapter 3, Putting Elasticsearch into Action, introduces Elasticsearch Query-DSL, various queries, and the data sorting techniques. You will also learn how to perform CRUD operations with Elasticsearch using Elasticsearch Python and Java clients.

Chapter 4, Aggregations for Analytics, is all about the Elasticsearch aggregation framework for building analytics on data. It provides many fundamental as well complex examples of data analytics that can be built using a combination of full-text search, term-based search, and multi level aggregations. The user will master the aggregation module of Elasticsearch by learning a complete set of practical code examples that are covered using Python and Java clients.

Chapter 5, Data Looks Better on Maps: Master Geo-Spatiality, discusses geo-data concepts and covers the rich geo-search functionalities offered by Elasticsearch including how to create mappings for geo-points and geo-shapes data, indexing documents, geo-aggregations, and sorting data based on geo-distance. It includes code examples for the most widely used geo-queries in both Python and Java.

Chapter 6, Document Relationships in NoSQL World, focuses on the techniques offered by Elasticsearch to handle relational data using nested and parent-child relationships and creating a schema for the same using real-world examples. The reader will also learn how to create mappings based on relational data and write code for indexing and querying data using Python and Java APIs.

Chapter 7, Different Methods of Search and Bulk Operations, covers the different types of search and bulk APIs that every programmer needs to know while developing applications and working with large data sets. You will learn examples of bulk processing, multi-searches, and faster data reindexing using both Python and Java, which will help you throughout your journey with Elasticsearch.

Chapter 8, Controlling Relevancy, discusses the most important aspect of search engines — relevancy. It covers the powerful scoring capabilities available in Elasticsearch and practical examples that show how you can control the scoring process according to your needs.
Chapter 9, *Cluster Scaling in Production Deployments*, shows how to create Elasticsearch clusters and configure different types of nodes with the right resource allocations. It also focuses on cluster scalability using the best practices in production environment.

Chapter 10, *Backups and Security*, focuses on the different mechanisms of creating data backups of an Elasticsearch cluster and restoring them back into the same or another cluster. A step-by-step guide to setting up NFS (Network File System) is also provided. Finally, you will learn about setting up Nginx to secure Elasticsearch and load balance requests.
Aggregations for Analytics

Elasticsearch is a search engine at the core but what makes it more usable is its ability to make complex data analytics in an easy and simple way. The volume of data is growing rapidly and companies want to perform analysis on data in real time. Whether it is log, real-time streaming of data, or static data, Elasticsearch works wonderfully in getting a summarization of data through its aggregation capabilities.

In this chapter, we will cover the following topics:

- Introducing the aggregation framework
- Metric and bucket aggregations
- Combining search, buckets, and metrics
- Memory pressure and implications

Introducing the aggregation framework

The aggregation functionality is completely different from search and enables you to ask sophisticated questions of the data. The use cases of aggregation vary from building analytical reports to getting real-time analysis of data and taking quick actions.

Also, despite being different in functionality, aggregations can operate along the usual search requests. Therefore, you can search or filter your data, and at the same time, you can also perform aggregation on the same datasets matched by search/filter criteria in a single request. A simple example can be to find the maximum number of hashtags used by users related to tweets that has crime in the text field. Aggregations enable you to calculate and summarize data about the current query on the fly. They can be used for all sorts of tasks such as dynamic counting of result values to building a histogram.
Aggregations come in two flavors: **metrics** and **buckets**.

- **Metrics**: Metrics are used to do statistics calculations, such as min, max, average, on a field of a document that falls into a certain criteria. An example of a metric can be to find the maximum count of followers among the user’s follower counts.

- **Buckets**: Buckets are simply the grouping of documents that meet a certain criteria. They are used to categorize documents, for example:
  - The category of loans can fall into the buckets of home loan or personal loan
  - The category of an employee can be either male or female

Elasticsearch offers a wide variety of buckets to categorize documents in many ways such as by days, age range, popular terms, or locations. However, all of them work on the same principle: **document categorization based on some criteria**.

The most interesting part is that bucket aggregations can be nested within each other. This means that a bucket can contain other buckets within it. Since each of the buckets defines a set of documents, one can create another aggregation on that bucket, which will be executed in the context of its parent bucket. For example, a country-wise bucket can include a state-wise bucket, which can further include a city-wise bucket.

In SQL terms, metrics are simply functions such as `MIN()`, `MAX()`, `SUM()`, `COUNT()`, and `AVG()`, where buckets group the results using `GROUP BY` queries.

### Aggregation syntax

Aggregation follows the following syntax:

```json
"aggregations" : {
    
    },"aggregations" : { ["sub_aggregation"]+ } ]?

},"<aggregation_name_2>" : { ... }]*
}
```
Let’s understand how the preceding structure works:

- **aggregations**: The aggregations objects (which can also be replaced with agg) in the preceding structure holds the aggregations that have to be computed. There can be more than one aggregation inside this object.

- **<aggregation_name>**: This is a user-defined logical name for the aggregations that are held by the aggregations object (for example, if you want to compute the average age of users in the index, it makes sense to give the name as `avg_age`). These logical names will also be used to uniquely identify the aggregations in the response.

- **<aggregation_type>**: Each aggregation has a specific type, for example, terms, sum, avg, min, and so on.

- **<aggregation_body>**: Each type of aggregation defines its own body depending on the nature of the aggregation (for example, an `avg` aggregation on a specific field will define the field on which the average will be calculated).

- **<sub_aggregation>**: The sub aggregations are defined on the bucketing aggregation level and are computed for all the buckets built by the bucket aggregation. For example, if you define a set of aggregations under the range aggregation, the sub aggregations will be computed for the range buckets that are defined.

Look at the following JSON structure to understand a more simple structure of aggregations:

```json
{
  "aggs": {
    "NAME1": {
      "AGG_TYPE": {},
      "aggs": {
        "NAME": {
          "AGG_TYPE": {}
        }
      }
    },
    "NAME2": {
      "AGG_TYPE": {}
    }
  }
}
```
Extracting values

Aggregations typically work on the values extracted from the aggregated document set. These values can be extracted either from a specific field using the field key inside the aggregation body or can also be extracted using a script.

While it's easy to define a field to be used to aggregate data, the syntax of using scripts needs some special understanding. The benefit of using scripts is that one can combine the values from more than one field to use as a single value inside an aggregation.

Using scripting requires much more computation power and slows down the performance on bigger datasets.

The following are the examples of extracting values from a script:

Extracting a value from a single field:

```json
{ "script" : "doc['field_name'].value" }
```

Extracting and combining values from more than one field:

```json
"script": "doc['author.first_name'].value + ' ' + doc['author.last_name'].value"
```

The scripts also support the use of parameters using the `param` keyword. For example:

```json
{
   "avg": {
      "field": "price",
      "script": {
         "inline": "_value * correction",
         "params": {
            "correction": 1.5
         }
      }
   }
}
```

The preceding aggregation calculates the average price after multiplying each value of the price field with 1.5, which is used as an inline function parameter.
Returning only aggregation results

Elasticsearch by default computes aggregations on a complete set of documents using the `match_all` query and returns 10 documents by default along with the output of the aggregation results.

If you do not want to include the documents in the response, you need to set the value of the `size` parameter to 0 inside your query. Note that you do not need to use the `from` parameter in this case. This is a very useful parameter because it avoids document relevancy calculation and the inclusion of documents in the response, and only returns the aggregated data.

Metric aggregations

As explained in the previous sections, metric aggregations allow you to find out the statistical measurement of the data, which includes the following:

- Computing basic statistics
  - Computing in a combined way: `stats` aggregation
  - Computing separately: `min`, `max`, `sum`, `value_count`, `aggregations`
- Computing extended statistics: `extended_stats` aggregation
- Computing distinct counts: `cardinality` aggregation

Metric aggregations are fundamentally categorized in two forms:

- **single-value metric**: `min`, `max`, `sum`, `value_count`, `avg`, and `cardinality` aggregations
- **multi-value metric**: `stats` and `extended_stats` aggregations
Computing basic stats
The basic statistics include: min, max, sum, count, and avg. These statistics can be computed in the following two ways and can only be performed on numeric fields.

Combined stats
All the stats mentioned previously can be calculated with a single aggregation query.

Python example
```python
query = {
    "aggs": {
        "follower_counts_stats": {
            "stats": {
                "field": "user.followers_count"
            }
        }
    }
}
res = es.search(index='twitter', doc_type='tweets', body=query)
print resp
```
The response would be as follows:
```
"aggregations": {
    "follower_counts_stats": {
        "count": 124,
        "min": 2,
        "max": 38121,
        "avg": 2102.814516129032,
        "sum": 260749
    }
}
```
In the preceding response, count is the total values on which the aggregation is executed.

- min is the minimum follower count of a user
- max is the maximum follower count of a user
- avg is the average count of followers
- Sum is the addition of all the followers count
Java example

In Java, all the metric aggregations can be created using the MetricsAggregationBuilder and AggregationBuilders classes. However, you need to import a specific package into your code to parse the results.

To build and execute a stats aggregation in Java, first do the following imports in the code:

```java
import org.elasticsearch.search.aggregations.metrics.stats.Stats;
```

Then build the aggregation in the following way:

```java
MetricsAggregationBuilder aggregation =
    AggregationBuilders.stats("follower_counts_stats")
    .field("user.followers_count");
```

This aggregation can be executed with the following code snippet:

```java
SearchResponse response = client.prepareSearch(indexName).
    setTypes(docType).setQuery(QueryBuilders.matchAllQuery())
    .addAggregation(aggregation)
    .execute().actionGet();
```

The stats aggregation response can be parsed as follows:

```java
Stats agg = sr.getAggregations().get("follower_counts_stats");
long min = agg.getMin();
long max = agg.getMax();
double avg = agg.getAvg();
long sum = agg.getSum();
long count = agg.getCount();
```

Computing stats separately

In addition to computing these basic stats in a single query, Elasticsearch provides multiple aggregations to compute them one by one. The following are the aggregation types that fall into this category:

- **value_count**: This counts the number of values that are extracted from the aggregated documents
- **min**: This finds the minimum value among the numeric values extracted from the aggregated documents
Aggregations for Analytics

- **max**: This finds the maximum value among the numeric values extracted from the aggregated documents
- **avg**: This finds the average value among the numeric values extracted from the aggregated documents
- **sum**: This finds the sum of all the numeric values extracted from the aggregated documents

To perform these aggregations, you just need to use the following syntax:

```json
{
    "aggs": {
        "aggaregation_name": {
            "aggrigation_type": {
                "field": "name_of_the_field"
            }
        }
    }
}
```

**Python example**

```python
query = {
    "aggs": {
        "follower_counts_stats": {
            "sum": {
                "field": "user.followers_count"
            }
        }
    },
    "size": 0
}
res = es.search(index='twitter', doc_type='tweets', body=query)
```

We used the **sum** aggregation type in the preceding query; for other aggregations such as **min, max, avg, and value_count**, just replace the type of aggregation in the query.

**Java example**

To perform these aggregations using the Java client, you need to follow this syntax:

```java
MetricsAggregationBuilder aggregation =
    AggregationBuilders
        .sum("follower_counts_stats")
        .field("user.followers_count");
```

Note that in the preceding aggregation, instead of sum, you just need to call the corresponding aggregation type to build other types of metric aggregations such as, min, max, count, and avg. The rest of the syntax remains the same.

For parsing the responses, you need to import the correct package according to the aggregation type. The following are the imports that you will need:

- **For min aggregation:**
  ```java
  import org.elasticsearch.search.aggregations.metrics.min.Min;
  ```
  The parsing response will be as follows:
  ```java
  Min agg = response.getAggregations().get("follower_counts_stats");
  double value = agg.getValue();
  ```

- **For max aggregation:**
  ```java
  import org.elasticsearch.search.aggregations.metrics.min.Max;
  ```
  The parsing response will be:
  ```java
  Max agg = response.getAggregations().get("follower_counts_stats");
  double value = agg.getValue();
  ```

- **For avg aggregation:**
  ```java
  import org.elasticsearch.search.aggregations.metrics.min.Avg;
  ```
  The parsing response will be this:
  ```java
  Avg agg = response.getAggregations().get("follower_counts_stats");
  double value = agg.getValue();
  ```

- **For sum aggregation:**
  ```java
  import org.elasticsearch.search.aggregations.metrics.min.Sum;
  ```
  This will be the parsing response:
  ```java
  Sum agg = response.getAggregations().get("follower_counts_stats");
  double value = agg.getValue();
  ```

Stats aggregations cannot contain sub aggregations. However, they can be a part of the sub aggregations of buckets.
Computing extended stats

The `extended_stats` aggregation is the extended version of `stats` aggregation and provides advanced statistics of the data, which include sum of square, variance, standard deviation, and standard deviation bounds.

So, if we hit the query with the `extended_stats` aggregation on the followers count field, we will get the following data:

```json
"aggregations": {
  "follower_counts_stats": {
    "count": 124,
    "min": 2,
    "max": 38121,
    "avg": 2102.814516129032,
    "sum": 260749,
    "sum_of_squares": 3334927837,
    "variance": 22472750.441402186,
    "std_deviation": 4740.543264374051,
    "std_deviation_bounds": {
      "upper": 11583.901044877135,
      "lower": -7378.272012619071
    }
  }
}
```

Python example

```python
query = {
  "aggs": {
    "follower_counts_stats": {
      "extended_stats": {
        "field": "user.followers_count"
      }
    }
  }
}, "size": 0
res = es.search(index='twitter', doc_type='tweets', body=query)
```
Java example

An extended aggregation is build using the Java client in the following way:

```java
MetricsAggregationBuilder aggregation =
    AggregationBuilders.
        extendedStats("agg_name")
    .field("user.follower_count");
```

To parse the response of the `extended_stats` aggregation in Java, you need to have the following `import` statement:

```java
import org.elasticsearch.search.aggregations.metrics.stats.extended.
    ExtendedStats;
```

Then the response can parsed in the following way:

```java
ExtendedStats agg = response.getAggregations().get("agg_name");
    double min = agg.getMin();
    double max = agg.getMax();
    double avg = agg.getAvg();
    double sum = agg.getSum();
    long count = agg.getCount();
    double stdDeviation = agg.getStdDeviation();
    double sumOfSquares = agg.getSumOfSquares();
    double variance = agg.getVariance();
```

**Finding distinct counts**

The count of a distinct value of a field can be calculated using the cardinality aggregation. For example, we can use this to calculate unique users:

```json
{
    "aggs": {
        "unique_users": {
            "cardinality": {
                "field": "user.screen_name"
            }
        }
    }
}
```
Aggregations for Analytics

The response will be as follows:

```
"aggregations": {
    "unique_users": {
        "value": 122
    }
}
```

Java example

Cardinality aggregation is built using the Java client in the following way:

```java
MetricsAggregationBuilder aggregation = AggregationBuilders.
    cardinality("unique_users")
    .field("user.screen_name");
```

To parse the response of the cardinality aggregation in Java, you need to have the following import statement:

```java
import org.elasticsearch.search.aggregations.metrics.cardinality.
Cardinality;
```

Then the response can parsed in the following way:

```java
Cardinality agg = response.getAggregations().get("unique_users");
long value = agg.getValue();
```

Bucket aggregations

Similar to metric aggregations, bucket aggregations are also categorized into two forms: Single buckets that contain only a single bucket in the response, and multi buckets that contain more than one bucket in the response.

The following are the most important aggregations that are used to create buckets:

- Multi bucket aggregations
  - Terms aggregation
  - Range aggregation
  - Date range aggregation
  - Histogram aggregation
  - Date histogram aggregation
Chapter 4

• Single bucket aggregation
  
  ° Filter-based aggregation

We will cover a few more aggregations such as nested and geo aggregations in subsequent chapters.

Buckets aggregation response formats are different from the response formats of metric aggregations. The response of a bucket aggregation usually comes in the following format:

```json
"aggregations": {

  "aggregation_name": {
    "buckets": [
      {
        "key": value,
        "doc_count": value
      },
      ......
    ]
  }
}
```

All the bucket aggregations can be created in Java using the AggregationBuilder and AggregationBuilders classes. You need to have the following classes imported inside your code for the same:

```java
org.elasticsearch.search.aggregations.AggregationBuilder;
org.elasticsearch.search.aggregations.AggregationBuilders;
```

Also, all the aggregation queries can be executed with the following code snippet:

```java
SearchResponse response = client.prepareSearch(indexName).setTypes(docType)
  .setQuery(QueryBuilders.matchAllQuery())
  .addAggregation(aggregation)
  .execute().actionGet();
```

The setQuery() method can take any type of Elasticsearch query, whereas the addAggregation() method takes the aggregation built using AggregationBuilder.
Terms aggregation

Terms aggregation is the most widely used aggregation type and returns the buckets that are dynamically built using one per unique value.

Let's see how to find the top 10 hashtags used in our Twitter index in descending order.

Python example

```python
query = {
    "aggs": {
        "top_hashtags": {
            "terms": {
                "field": "entities.hashtags.text",
                "size": 10,
                "order": {
                    "_term": "desc"
                }
            }
        }
    }
}
```

In the preceding example, the size parameter controls how many buckets are to be returned (defaults to 10) and the order parameter controls the sorting of the bucket terms (defaults to asc):

```python
res = es.search(index='twitter', doc_type='tweets', body=query)
```

The response would look like this:

```json
"aggregations": {
    "top_hashtags": {
        "doc_count_error_upper_bound": 0,
        "sum_other_doc_count": 44,
        "buckets": [
            {
                "key": "politics",
                "doc_count": 2
            },
            .............
        ]
    }
}
```
Java example

Terms aggregation can be built as follows:

```java
AggregationBuilder aggregation =
    AggregationBuilders.terms("agg").field(fieldName).
        size(10);
```

Here, `agg` is the aggregation bucket name and `fieldName` is the field on which the aggregation is performed.

The response object can be parsed as follows:

To parse the terms aggregation response, you need to import the following class:

```java
import org.elasticsearch.search.aggregations.bucket.terms.Terms;
```

Then, the response can be parsed with the following code snippet:

```java
Terms screen_names = response.getAggregations().get("agg");
    for (Terms.Bucket entry : screen_names.getBuckets()) {
        entry.getKey();  // Term
        entry.getDocCount(); // Doc count
    }
```

Range aggregation

With range aggregation, a user can specify a set of ranges, where each range represents a bucket. Elasticsearch will put the document sets into the correct buckets by extracting the value from each document and matching it against the specified ranges.

Python example

```python
query = "aggs": {
    "status_count_ranges": {
        "range": {
            "field": "user.statuses_count",
            "ranges": [
                {
                    "to": 50
                },
                {
                    "from": 50,
                    "to": 100
                }
            ]
        }
    }
}
```
The range aggregation always discards the `to` value for each range and only includes the `from` value.

The response for the preceding query request would look like this:

```json
"aggregations": {
  "status_count_ranges": {
    "buckets": [
      {
        "key": "*-50.0",
        "to": 50,
        "to_as_string": "50.0",
        "doc_count": 3
      },
      {
        "key": "50.0-100.0",
        "from": 50,
        "from_as_string": "50.0",
        "to": 100,
        "to_as_string": "100.0",
        "doc_count": 3
      }
    ]
  }
}
```

Java example

**Building range aggregation:**

```java
AggregationBuilder aggregation =
    AggregationBuilders
        .range("agg")
        .field(fieldName)
        .addUnboundedTo(1)   // from -infinity to 1 (excluded)
        .addRange(1, 100)   // from 1 to 100 (excluded)
        .addUnboundedFrom(100); // from 100 to +infinity
```
Here, \texttt{agg} is the aggregation bucket name and \texttt{fieldName} is the field on which the aggregation is performed. The \texttt{addUnboundedTo} method is used when you do not specify the \texttt{from} parameter and the \texttt{addUnboundedFrom} method is used when you don't specify the \texttt{to} parameter.

**Parsing the response**

To parse the range aggregation response, you need to import the following class:

```java
import org.elasticsearch.search.aggregations.bucket.range.Range;
```

Then, the response can be parsed with the following code snippet:

```java
Range agg = response.getAggregations().get("agg");
for (Range.Bucket entry : agg.getBuckets()) {
    String key = entry.getKeyAsString();       // Range as key
    Number from = (Number) entry.getFrom();   // Bucket from
    Number to = (Number) entry.getTo();       // Bucket to
    long docCount = entry.getDocCount();      // Doc count
}
```

**Date range aggregation**

The date range aggregation is dedicated for date fields and is similar to range aggregation. The only difference between range and date range aggregation is that the latter allows you to use a date math expression inside the from and to fields.

The following table shows an example of using math operations in Elasticsearch. The supported time units for the math operations are: \texttt{y} (year), \texttt{M} (month), \texttt{w} (week), \texttt{d} (day), \texttt{h} (hour), \texttt{m} (minute), and \texttt{s} (second):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Now}</td>
<td>Current time</td>
</tr>
<tr>
<td>\texttt{Now+1h}</td>
<td>Current time plus 1 hour</td>
</tr>
<tr>
<td>\texttt{Now-1M}</td>
<td>Current time minus 1 month</td>
</tr>
<tr>
<td>\texttt{Now+1h+1m}</td>
<td>Current time plus 1 hour plus one minute</td>
</tr>
<tr>
<td>\texttt{Now+1h/d}</td>
<td>Current time plus 1 hour rounded to the nearest day</td>
</tr>
<tr>
<td>\texttt{2016-01-01 +1M/d}</td>
<td>2016-01-01 plus 1 month rounded to the nearest day</td>
</tr>
</tbody>
</table>

**Python example**

```python
query = {
    "aggs": {
        "tweets_creation_interval": {
...
```
Aggregations for Analytics

```
"range": {
    "field": "created_at",
    "format": "yyyy",
    "ranges": [
        {
            "to": 2000
        },
        {
            "from": 2000,
            "to": 2005
        },
        {
            "from": 2005
        }
    ]
}, "size": 0
}
res = es.search(index='twitter', doc_type='tweets', body=query)
print res
```

Java example

**Building date range aggregation:**

```java
AggregationBuilder aggregation =
    AggregationBuilders.dateRange("agg")
        .field(fieldName)
        .format("yyyy")
        .addUnboundedTo("2000") // from -infinity to 2000 (excluded)
        .addRange("2000", "2005") // from 2000 to 2005 (excluded)
        .addUnboundedFrom("2005"); // from 2005 to +infinity
```

Here, agg is the aggregation bucket name and fieldName is the field on which the aggregation is performed. The addUnboundedTo method is used when you do not specify the from parameter and the addUnboundedFrom method is used when you don't specify the to parameter.

**Parsing the response:**

To parse the date range aggregation response, you need to import the following class:

```java
import org.elasticsearch.search.aggregations.bucket.range.Range;
import org.joda.time.DateTime;
```
Then, the response can be parsed with the following code snippet:

```java
Range agg = response.getAggregations().get("agg");
for (Range.Bucket entry : agg.getBuckets()) {
    String key = entry.getKeyAsString(); // Date range as key
    DateTime fromAsDate = (DateTime) entry.getFrom(); // Date bucket from as a Date
    DateTime toAsDate = (DateTime) entry.getTo(); // Date bucket to as a Date
    long docCount = entry.getDocCount(); // Doc count
}
```

### Histogram aggregation

A histogram aggregation works on numeric values extracted from documents and creates fixed-sized buckets based on those values. Let's see an example for creating buckets of a user's favorite tweet counts:

**Python example**

```python
query = {
    "aggs": {
        "favorite_tweets": {
            "histogram": {
                "field": "user.favourites_count",
                "interval": 20000
            }
        }
    },
    "size": 0
}
res = es.search(index='twitter', doc_type='tweets', body=query)
for bucket in res['aggregations']['favorite_tweets']['buckets']:
    print bucket['key'], bucket['doc_count']
```

The response for the preceding query will look like the following, which says that 114 users have favorite tweets between 0 to 20000 and 8 users have more than 20000 as their favorite tweets:

```
"aggregations": {
    "favorite_tweets": {
        "buckets": [
            {
                "key": 0,
                "doc_count": 114
            }
        ]
    }
```

---

**Chapter 4**

[89]
Aggregations for Analytics

```json
{
    "key": 20000,
    "doc_count": 8
}
}
```

While executing the histogram aggregation, the values of the documents are rounded off and they fall into the closest bucket; for example, if the favorite tweet count is 72 and the bucket size is set to 5, it will fall into the bucket with the key 70.

Java example

Building histogram aggregation:

```java
AggregationBuilder aggregation = AggregationBuilders.histogram("agg")
    .field(fieldName)
    .interval(5);
```

Here, `agg` is the aggregation bucket name and `fieldName` is the field on which aggregation is performed. The `interval` method is used to pass the interval for generating the buckets.

Parsing the response:

To parse the histogram aggregation response, you need to import the following class:

```java
import org.elasticsearch.search.aggregations.bucket.histogram.Histogram;
```

Then, the response can be parsed with the following code snippet:

```java
Range agg = response.getAggregations().get("agg");
for (Histogram.Bucket entry : agg.getBuckets()) {
    Long key = (Long) entry.getKey();       // Key
    long docCount = entry.getDocCount();    // Doc coun
}
```
Date histogram aggregation

Date histogram is similar to the histogram aggregation but it can only be applied to date fields. The difference between the two is that date histogram allows you to specify intervals using date/time expressions.

The following values can be used for intervals:

- year, quarter, month, week, day, hour, minute, and second

You can also specify fractional values, such as 1h (1 hour), 1m (1 minute) and so on.

Date histograms are mostly used to generate time-series graphs in many applications.

Python example

```python
query = {
    "aggs": {
        "tweet_histogram": {
            "date_histogram": {
                "field": "created_at",
                "interval": "hour"
            }
        }
    },
    "size": 0
}
```

The preceding aggregation will generate an hourly-based tweet timeline on the field, created_at:

```python
res = es.search(index='twitter', doc_type='tweets', body=query)
for bucket in res['aggregations']['tweet_histogram']['buckets']:
    print bucket['key'], bucket['key_as_string'], bucket['doc_count']
```

Java example

Building date histogram aggregation:

```java
AggregationBuilder aggregation =
    AggregationBuilders.histogram("agg")
        .field(fieldName)
        .interval(DateHistogramInterval.YEAR);
```
Aggregations for Analytics

Here, agg is the aggregation bucket name and fieldname is the field on which the aggregation is performed. The interval method is used to pass the interval to generate buckets. For interval in days, you can do this:

```java
DateHistogramInterval.days(10)
```

Parsing the response:

To parse the date histogram aggregation response, you need to import the following class:

```java
import org.elasticsearch.search.aggregations.bucket.histogram.
DateHistogramInterval;
```

The response can be parsed with this code snippet:

```java
Histogram agg = response.getAggregations().get("agg");
for (Histogram.Bucket entry : agg.getBuckets()) {
    DateTime key = (DateTime) entry.getKey();    // Key
    String keyAsString = entry.getKeyAsString(); // Key as String
    long docCount = entry.getDocCount();         // Doc count
}
```

Filter-based aggregation

Elasticsearch allows filters to be used as aggregations too. Filters preserve their behavior in the aggregation context as well and are usually used to narrow down the current aggregation context to a specific set of documents. You can use any filter such as range, term, geo, and so on.

To get the count of all the tweets done by the user, d_bharvi, use the following code:

Python example

```python
query = {
    "aggs": {
        "screenname_filter": {
            "filter": {
                "term": {
                    "user.screen_name": "d_bharvi"
                }
            }
        }
    },
    "size": 0
}
```
In the preceding request, we have used a term filter to narrow down the bucket of tweets done by a particular user:

```python
res = es.search(index='twitter', doc_type='tweets', body=query)
for bucket in res['aggregations']['screename_filter']['buckets']:
    print(bucket['doc_count'])
```

The response would look like this:

```
"aggregations": {
    "screename_filter": {
        "doc_count": 100
    }
}
```

Java example

**Building filter-based aggregation:**

```java
AggregationBuilder aggregation =
    AggregationBuilders.filter("agg")
        .filter(QueryBuilders.termQuery("user.screen_name ", "d_bharvi"));
```

Here, agg is the aggregation bucket name under the first filter method and the second filter method takes a query to apply the filter.

**Parsing the response:**

To parse a filter-based aggregation response, you need to import the following class:

```java
import org.elasticsearch.search.aggregations.bucket.histogram.DateHistogramInterval;
```

The response can be parsed with the following code snippet:

```java
Filter agg = response.getAggregations().get("agg");
agg.getDocCount(); // Doc count
```
Combining search, buckets, and metrics

We can always combine searches, filters bucket aggregations, and metric aggregations to get a more and more complex analysis. Until now, we have seen single levels of aggregations; however, as explained in the aggregation syntax section earlier, an aggregation can contain multiple levels of aggregations within. However, metric aggregations cannot contain further aggregations within themselves. Also, when you run an aggregation, it is executed on all the documents in the index for a document type if specified on a match_all query context, but you can always use any type of Elasticsearch query with an aggregation. Let's see how we can do this in Python and Java clients.

Python example

```python
query = {
    "query": {
        "match": {
            "text": "crime"
        }
    },
    "aggs": {
        "hourly_timeline": {
            "date_histogram": {
                "field": "created_at",
                "interval": "hour"
            },
            "aggs": {
                "top_hashtags": {
                    "terms": {
                        "field": "entities.hashtags.text",
                        "size": 1
                    },
                    "aggs": {
                        "top_users": {
                            "terms": {
                                "field": "user.screen_name",
                                "size": 1
                            },
                            "aggs": {
                                "average_tweets": {
                                    "avg": {
                                        "field": "user.statuses_count"
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```
res = es.search(index='twitter', doc_type='tweets', body=query)

Parsing the response data:

    for timeline_bucket in res['aggregations']['hourly_timeline']['buckets']:
        print 'time range', timeline_bucket['key_as_string']
        print 'tweet count', timeline_bucket['doc_count']
        for hashtag_bucket in timeline_bucket['top_hashtags']['buckets']:
            print 'hashtag key', hashtag_bucket['key']
            print 'hashtag count', hashtag_bucket['doc_count']
            for user_bucket in hashtag_bucket['top_users']['buckets']:
                print 'screen_name', user_bucket['key']
                print 'count', user_bucket['doc_count']
                print 'average tweets', user_bucket['average_tweets']
        ['value']

And you will find the output as below:

time_range 2015-10-14T10:00:00.000Z
tweet_count 1563

hashtag_key crime
hashtag_count 42

    screen_name andresenior
    count 2
    average_tweets 9239.0

Understanding the response in the context of our search of the term crime in a text field:

- **time_range**: The key of the daywise_timeline bucket
- **tweet_count**: The number of tweets happening per hour
- **hashtag_key**: The name of the hashtag used by users within the specified time bucket
Aggregations for Analytics

- **hashtag_count**: The count of each hashtag within the specified time bucket
- **screen_name**: The screen name of the user who has tweeted using that hashtag
- **count**: The number of times that user tweeted using a corresponding hashtag
- **average_tweets**: The average number of tweets done by users in their lifetime who have used this particular hashtag

Java example

Writing multilevel aggregation queries (as we just saw) in Java seems quite complex, but once you learn the basics of structuring aggregations, it becomes fun.

Let's see how we write the previous query in Java:

**Building the query using QueryBuilder**:

```java
QueryBuilder query = QueryBuilders.matchQuery("text", "crime");
```

**Building the aggregation**:

The syntax for a multilevel aggregation in Java is as follows:

```java
AggregationBuilders
    .aggType("aggs_name")
    //aggregation_definition
    .subAggregation(AggregationBuilders
        .aggType("aggs_name")
        //aggregation_definition
        .subAggregation(AggregationBuilders
            .aggType("aggs_name")
            //aggregation_definition
            .subAggregation(AggregationBuilders
                .aggType("aggs_name")
                //aggregation_definition......

You can relate the preceding syntax with the aggregation syntax you learned in the beginning of this chapter.

The exact aggregation for our Python example will be as follows:

```java
AggregationBuilder aggregation =
    AggregationBuilders
        .dateHistogram("hourly_timeline")
        .field("@timestamp")
        .interval(DateHistogramInterval.YEAR)
        .subAggregation(AggregationBuilders
            .terms("top_hashtags")
            .field("entities.hashtags.text")
            .subAggregation(AggregationBuilders...)
```
Let's execute the request by combining the query and aggregation we have built:

```java
SearchResponse response = client.prepareSearch(indexName).
    setTypes(docType)
    .setQuery(query).addAggregation(aggregation)
    .setSize(0)
    .execute().actionGet();
```

**Parsing multilevel aggregation responses:**

Since multilevel aggregations are nested inside each other, you need to iterate accordingly to parse each level of aggregation response in loops.

The response for our request can be parsed with the following code:

```java
//Get first level of aggregation data
Histogram agg = response.getAggregations().get("hourly_timeline");
//for each entry of hourly histogram
for (Histogram.Bucket entry : agg.getBuckets()) {
    DateTime key = (DateTime) entry.getKey();
    String keyAsString = entry.getKeyAsString();
    long docCount = entry.getDocCount();
    System.out.println(key);
    System.out.println(docCount);
}

//Get second level of aggregation data
Terms topHashtags = entry.getAggregations().get("top_hashtags");
//for each entry of top hashtags
for (Terms.Bucket hashTagEntry : topHashtags.getBuckets()) {
    String hashtag = hashTagEntry.getKey().toString();
    long hashtagCount = hashTagEntry.getDocCount();
    System.out.println(hashtag);
    System.out.println(hashtagCount);
}

//Get 3rd level of aggregation data
Terms topUsers = hashTagEntry.getAggregations().get("top_users");
//for each entry of top users
for (Terms.Bucket usersEntry : topUsers.getBuckets()) {
    //...
Aggregations for Analytics

```java
String screenName = usersEntry.getKey().toString();
long userCount = usersEntry.getDocCount();
System.out.println(screenName);
System.out.println(userCount);

// Get 4th level of aggregation data
Avg average_status_count = usersEntry.getAggregations().get("average_status_count");
double max = average_status_count.getValue();
System.out.println(max);
```

As you saw, building these types of aggregations and going for a drill down on data sets to do complex analytics can be fun. However, one has to keep in mind the pressure on memory that Elasticsearch bears while doing these complex calculations. The next section covers how we can avoid these memory implications.

**Memory pressure and implications**

Aggregations are awesome! However, they bring a lot of memory pressure on Elasticsearch. They work on an in-memory data structure called **fielddata**, which is the biggest consumer of HEAP memory in a Elasticsearch cluster. Fielddata is not only used for aggregations, but also used for sorting and scripts. The in-memory fielddata is slow to load, as it has to read the whole inverted index and un-invert it. If the fielddata cache fills up, old data is evicted causing heap churn and bad performance (as fielddata is reloaded and evicted again.)

The more unique terms exist in the index, the more terms will be loaded into memory and the more pressure it will have. If you are using an Elasticsearch version below 2.0.0 and above 1.0.0, then you can use the `doc_values` parameter inside the mapping while creating the index to avoid the use of fielddata using the following syntax:

```json
PUT /index_name/_mapping/index_type
{
  "properties": {
    "field_name": {
      "type": "string",
      "index": "not_analyzed",
      "doc_values": true
    }
  }
}
```
doc_values have been enabled by default from Elasticsearch version 2.0.0 onwards.

The advantages of using doc_values are as follows:

- Less heap usage and faster garbage collections
- No longer limited by the amount of fielddata that can fit into a given amount of heap—instead the file system caches can make use of all the available RAM
- Fewer latency spikes caused by reloading a large segment into memory

The other important consideration to keep in mind is not to have a huge number of buckets in a nested aggregation. For example, finding the total order value for a country during a year with an interval of one week will generate 100*51 buckets with the sum value. It is a big overhead that is not only calculated in data nodes, but also in the co-ordinating node that aggregates them. A big JSON also gives problems on parsing and loading on the "frontend". It will easily kill a server with wide aggregations.

Summary

In this chapter, we learned about one of the most powerful features of Elasticsearch, that is, aggregation frameworks. We went through the most important metric and bucket aggregations along with examples of doing analytics on our Twitter dataset with Python and Java API.

This chapter covered many fundamental as well complex examples of the different facets of analytics, which can be built using a combination of full-text searches, term-based searches, and multilevel aggregations. Elasticsearch is awesome for analytics but one should always keep in mind the memory implications, which we covered in the last section of this chapter, to avoid the over killing of nodes.

In the next chapter, we will learn to work with geo spatial data in Elasticsearch and we will also cover analytics with geo aggregations.
**Where to buy this book**

You can buy Elasticsearch Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.