INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Datenanalyse und Visualisierung mit Hilfe des ELK-Stacks am Beispiel League of Legends

Philipp Johannes Roß

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Datenanalyse und Visualisierung mit Hilfe des ELK-Stacks am Beispiel League of Legends

Philipp Johannes Roß

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Dr. Nils gentschen Felde Jan Schmidt Angelika Heimann (xdi360)
Abgabetermin:	24. März 2017

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 24. März 2017

.....
(Unterschrift des Kandidaten)

Abstract

Die Analyse und Visualisierung großer Datenmengen ist eine schwierige Aufgabe. Der *ELK-Stack* (Elasticsearch, Logstash, Kibana) bietet eine Möglichkeit, Daten zu sammeln, zu speichern und zu visualisieren. In dieser Arbeit werden die Eigenschaften und Funktionen des *ELK-Stacks* untersucht, um zu überprüfen, ob der *ELK-Stack* eine geeignete Lösung für die Problematik der Datenanalyse und Visualisierung darstellt. Dies geschieht durch die Umsetzung eines Versuchsaufbaus, bei dem der *ELK-Stack* verwendet wird. Dabei werden Daten aus dem Videospiel *League of Legends* genutzt, da diese durch ihre Struktur, Verfügbarkeit und Analysierbarkeit als Anwendungsbeispiel zur Datenanalyse und Visualisierung geeignet sind. Vor der Umsetzung werden zunächst die Grundlagen der einzelnen Komponenten des *ELK-Stacks* und der Datenquelle (Riot *API*) vermittelt. Jeder der drei Bestandteile besitzt eine andere Aufgabe. Logstash ist dafür zuständig, Daten aus einer Quelle auszulesen, aufzubereiten und anschließend an Elasticsearch weiterzuleiten. Elasticsearch ermöglicht die verteilte Speicherung der Daten und stellt eine Such- und Analytikmaschine bereit, die von Kibana genutzt wird, um die Erstellung von Visualisierungen und Kennzahlen zu ermöglichen. Mithilfe eines entworfenen Anforderungskataloges wird der Versuchsaufbau praktisch umgesetzt, indem die einzelnen Komponenten des *ELK-Stacks* implementiert werden. Die Evaluation der Ergebnisse führt zu dem Schluss, dass der *ELK-Stack* eine grundsätzlich geeignete Methodik zur Analyse und Visualisierung von Daten bietet, die allerdings bei Aspekten wie der Entwicklung eigener Visualisierungen noch verbessert werden kann.

Inhaltsverzeichnis

1. Einleitung **1**

2. Grundlagen **3**
2.1. ELK-Stack . . . 3
2.1.1. Logstash . . . 3
2.1.2. Elasticsearch . . . 9
2.1.3. Kibana . . . 17
2.2. League of Legends und Riot API . . . 23
2.2.1. League of Legends . . . 23
2.2.2. Riot API . . . 27
2.2.3. Eignung für den ELK-Stack . . . 29

3. Implementierung **31**
3.1. Entwurf des Versuchsaufbaus . . . 31
3.1.1. Anforderungen . . . 31
3.1.2. Aufbau . . . 33
3.1.3. Zusammenfassung . . . 33
3.2. ELK-Stack . . . 34
3.2.1. Konfiguration von Logstash . . . 34
3.2.2. Einrichten von Elasticsearch . . . 37
3.2.3. Erstellen von Visualisierungen mit Kibana . . . 38

4. Fazit **45**
4.1. Bewertung der Ergebnisse . . . 45
4.1.1. Funktionale Anforderungen . . . 45
4.1.2. Nichtfunktionale Anforderungen . . . 46
4.1.3. Zusammenfassung . . . 47
4.2. Ausblick . . . 48
4.2.1. Anwendbarkeit des ELK-Stacks . . . 48
4.2.2. Weitere Ansatzpunkte . . . 48

Abbildungsverzeichnis **49**

Literaturverzeichnis **51**

1. Einleitung

Der Datenverkehr im Internet nimmt immer größere Ausmaße an, laut Cisco wird er 2020 2,3 Zettabyte pro Jahr betragen [Cis16]. Mit den immer größer werdenden Mengen an Daten ist auch der Bereich *Big Data* in den letzten Jahren immer mehr gewachsen. Somit hat sich die Frage entwickelt, wie man diese großen Mengen sammeln, speichern und aus diesen unstrukturierten Daten wertvolle Erkenntnisse gewinnen kann.
Auch in der Videospielbranche fallen große Datenmengen an. So ist *League of Legends* das momentan meist gespielte Videospiel und wird von über 67 Millionen monatlichen Spielern gespielt [Tas14]. Mit steigender Popularität hat sich eine Szene mit professionellen Teams entwickelt, in der bei Turnieren nicht nur beträchtliche Summen gewonnen, sondern auch Millionen von Zuschauern vor die Bildschirme gelockt werden (2015 waren es bei der Weltmeisterschaft im Durchschnitt 4,2 Millionen Zuschauer pro Spiel [Mag15]). Bei jedem Spiel fallen Daten an, die mit der frei verfügbaren Riot *API* abgerufen werden können. Diese Daten müssen allerdings strukturiert und analysiert werden, damit Analysten oder Kommentatoren mithilfe von Kennzahlen oder Visualisierungen entscheiden können, was zu Sieg oder Niederlage geführt hat, um daraus Rückschlüsse für zukünftige Spiele abzuleiten.
Der *ELK-Stack* stellt eine *Open-Source* Antwort auf die Frage nach der Erkenntnisgewinnung aus großen Datenmengen dar. Der Stack soll die Analyse und Visualisierung von Daten ermöglichen und besteht aus drei aufeinander aufbauenden Programmen, Logstash, Elasticsearch und Kibana. Logstash stellt eine Pipeline dar, die Daten auslesen, aufbereiten und anschließend an Elasticsearch weiterleiten kann. Elasticsearch ermöglicht die verteilte Speicherung, Volltextsuche und Analyse der Daten. In Kibana können die Funktionen von Elasticsearch genutzt werden, um Visualisierungen und Kennzahlen zu erstellen.
Ziel der Arbeit ist es, zu prüfen, ob der *ELK-Stack* eine geeignete Methode zur Analyse und Visualisierung von Daten darstellt. Um dies zu untersuchen wird ein Versuchsaufbau implementiert, der alle Bestandteile des *ELK-Stacks* unter Verwendung von Beispieldaten aus *League of Legends* nutzt. Dabei soll Logstash Daten mit der Riot *API* abrufen, diese aufbereiten und an Elasticsearch weiterleiten. Elasticsearch speichert diese Daten und stellt Funktionen zur Analyse bereit. In Kibana sollen diese Funktionen genutzt und aus den Daten ein *Dashboard* mit Visualisierungen und Kennzahlen erstellt werden. Die Ergebnisse werden abschließend anhand eines zuvor aufgestellten Anforderungskatalog bewertet, um die Eignung des *ELK-Stack* zu beurteilen.
Um das Ziel zu erreichen folgt diese Arbeit einer bestimmten Struktur: Zu Beginn werden im Kapitel 2 die Grundlagen von Logstash, Elasticsearch und Kibana vermittelt, die für die Implementierung notwendig sind. Außerdem wird ein Überblick über das Spiel *League of Legends*, dessen Eignung als Anwendungsbeispiel und die Riot *API* gegeben. Kapitel 3 dokumentiert die Implementierung des Versuchsaufbaus. Dafür wird zunächst ein Entwurf erstellt, der die Anforderungen an den Versuchsaufbau behandelt. Danach werden die einzelnen Bestandteile des *ELK-Stacks* unter Verwendung der Daten aus dem Spiel *League of Legends* implementiert. In Kapitel 4 wird ein Fazit gezogen, indem die Ergebnisse unter Einbezug des Anforderungskataloges bewertet werden. Abschließend wird ein Ausblick über die Anwendbarkeit des *ELK-Stacks* gegeben. Darüber hinaus werden Ansatzpunkte für die weitere Behandlung der Thematik angeführt.

2. Grundlagen

Dieses Kapitel beschäftigt sich mit den Grundlagen, die für die Implementierung des Versuchsaufbau benötigt werden. Es handelt sich hier lediglich um grundlegende Informationen über die einzelne Bestandteile des *ELK-Stacks* und nicht um eine vollständige Dokumentation. Auch wird hier das grundlegende Wissen über das Videospiel *League of Legends* vermittelt und behandelt, warum die Daten von *League of Legends* als Anwendungsbeispiel für den *ELK-Stack* geeignet sind.

2.1. ELK-Stack

Die Aufgabe des *ELK-Stacks* ist die Sammlung, Speicherung und Analyse bzw. Visualisierung von Daten. Jede dieser Aufgaben wird durch eine andere Komponente realisiert, nämlich Elasticsearch, Logstash und Kibana (daher der Name *ELK-Stack*). Über diese Bestandteile wird in den folgenden Kapitel ein Grundlagenwissen vermittelt. Die Grafik 2.1 zeigt den Aufbau des *Stacks* und die Interaktionen zwischen den Komponenten.

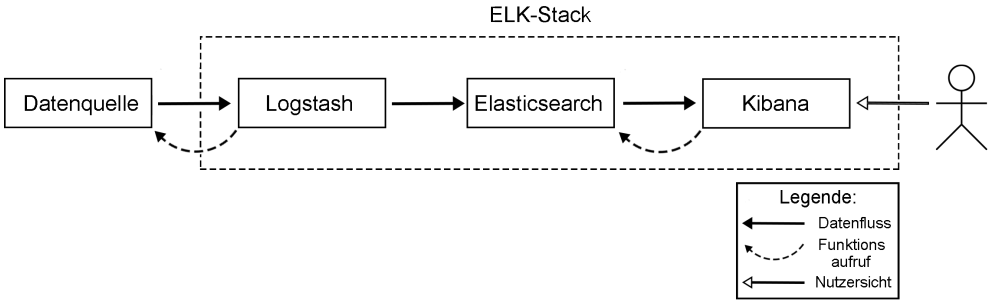

Abbildung 2.1.: *ELK-Stack*

Im Gegensatz zur Abkürzung ist der Datenfluss im *ELK-Stack* nicht Elasticsearch ⇒ Logstash ⇒ Kibana, sondern Logstash ⇒ Elasticsearch ⇒ Kibana.

2.1.1. Logstash

Das folgende Kapitel handelt von Logstash und basiert auf den *Logstash Reference* Dokumenten [ela16c]. Logstash, wie auch Elasticsearch und Kibana, ist ein *Open-Source* Project der Firma Elastic. Ursprünglich für das Sammeln von Logdateien konzipiert, ermöglicht Logstash nun das Sammeln von Daten verschiedener Typen. Diese Daten können anschließend gefiltert, transformiert oder auch nur auf das Wesentliche reduziert werden, um danach unter

anderem zur Analyse (z. B. Elasticsearch), Überwachung (z. B. Nagios) oder Archivierung (z. B. Google Cloud Storage) weitergeleitet werden zu können.

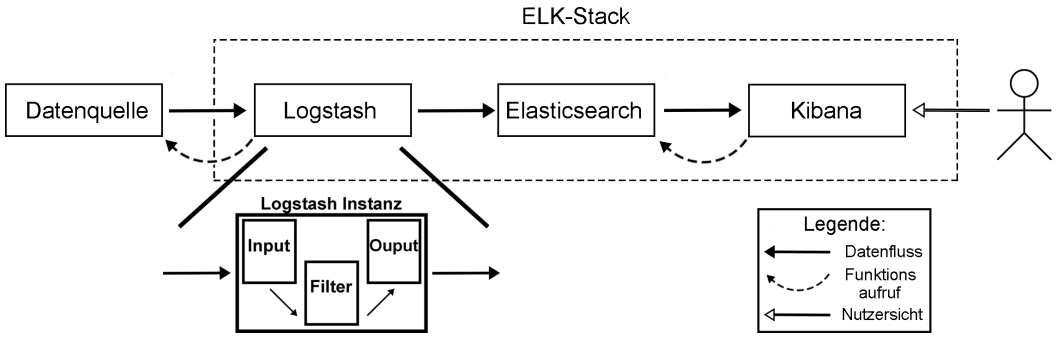

Abbildung 2.2.: Rolle von Logstash im *ELK-Stack* [ela16d]

Allgemeines

Die einzelnen Stationen der Daten im Weg durch die Pipeline, die in der Abbildung 2.2 zu sehen sind, werden in diesem Kapitel in der gleichen Reihenfolge behandelt, wie sie in einer Konfigurationsdatei vorkommen. Logstash Konfigurationsdateien sind im JSON Format und befinden sich normalerweise im Verzeichnis `/etc/logstash/conf.d`.
Eine Konfigurationsdatei ist folgendermaßen strukturiert:

```
input {
  [...]
}
# Filter sind optional
filter {
  [...]
}

output {
  [...]
}
```

Listing 2.1: Struktur einer Logstash Konfigurationsdatei

Dabei sind die Teile *Input* und *Output* zwingend notwendig, während der Abschnitt *Filter* nur optional ist. *Input* ist dafür zuständig, Daten aus verschiedensten Quellen in bestimmten Intervallen auszulesen. Diese Daten können anschließend im *Filter* Abschnitt strukturiert, umgewandelt oder um Informationen erweitert bzw. reduziert werden. Abschließend müssen im *Output* Block eine oder mehrere Zieldestinationen für die Daten konfiguriert werden, um die Daten an diese weiterzuleiten. Jeder dieser 3 Bestandteile benötigt verschiedene Funktionen (englisch *Plugins*), die je nach Bedarf zu wählen und zu konfigurieren sind. Dabei können mehrere Funktionen hintereinander verwendet werden, die in der Reihenfolge der Nennung angewendet werden.

Unterstützte Datentypen Funktionen in Logstash benötigen oft Daten eines bestimmten Typs wie Listen oder Zahlen, um beispielsweise mehrere Dateien auszulesen. Folgende Datentypen werden in Logstash unterstützt:

Listen Ein Liste kann aus einen oder mehreren Werten bestehen.
Beispiel:

```
path => [ "C:\Users\X\matches3.json",
          "C:\Users\X\matches2.json"]
path => "C:\Users\X\matches1.json"
```

Listing 2.2: Listen in Logstash

Boolean Dieser Datentyp dient dazu, einen logischen Wert zu speichern, der entweder wahr (`true`) oder falsch (`false`) sein muss.
Beispiel:

```
keepalive = true
```

Listing 2.3: Boolesche Variablen

Bytes Bytes werden durch ein Stringfeld repräsentiert. Die Verwendung von *SI*-Präfixen (k,M,G,T,P,E,Z,Y) und von *IEC*-Präfixen (Ki,Mi,Gi,Ti,Pi,Ei,Zi,Yi) ist hierbei möglich [ela16c, Codec Plugins, Bytes]. Das Feld ist *case* insensitiv und falls nicht weiter spezifiziert, wird die Einheit des Stringfeldes als Anzahl der Bytes interpretiert.
Beispiel:

```
bytes => "1234"
bytes => "1MiB"
bytes => "100kib"
bytes => "1 gb"
```

Listing 2.4: Darstellung von Bytes

Codec *Codecs* sind hilfreich, um Daten zu kodieren bzw. dekodieren. Sie werden in *Input* und *Output* verwendet. *Inputcodecs* dekodieren die Daten, bevor sie den *Inputbereich* betreten. *Outputcodecs* kodieren die Daten vor Verlassen des *Outputbereiches*.
Beispiel:

```
codec => "json"
```

Listing 2.5: Codecs

In der Logstash Dokumentation ist eine vollständige Liste mit *Codecs* verfügbar [ela16c, Codec Plugins].

Hash Hier können Feldern Werte zugeordnet werden mit dem Format `"Feld1"⇒"Wert1"`. Mehrere Einträge werden hierbei durch Leerzeichen getrennt und nicht mit Kommata.
Beispiel:

```
match => {
  "Feld1" => "Wert1"
  "Feld2" => "Wert2"
}
```

Listing 2.6: Zuordnung von Werten

Zahlen Zahlen müssen Integer- oder Fließkommazahlen sein.
Beispiel:

```
port => 9200
```

Listing 2.7: Repräsentation von Zahlen

Pfade Pfade werden in Logstash durch Strings dargestellt und müssen gültige Systempfade sein.
Beispiel:

```
path => "C:\Users\X\matches1.json"
```

Listing 2.8: Pfade in Logstash

Strings Strings werden in einfachen oder doppelten Anführungszeichen gerahmt. Anführungszeichen, die innerhalb des Strings vorkommen, müssen mit einem Schrägstrich maskiert werden.
Beispiel:

```
field => "entries"
field => 'ent\'ries'
```

Listing 2.9: Strings in Logstash

Nun werden die 3 Komponenten einer Logstashkonfiguration in den Kapiteln A, B und C behandelt.

A. Input

Logstash ist in der Lage Daten verschiedenster Formate aus verschiedensten Quellen auszulesen. Dies geschieht im *Input* Bereich mithilfe zahlreicher Funkionen, auch *Plugins* genannt, die je nach Art der Quelle konfiguriert werden müssen. An dieser Stelle werden nur ein paar der zahlreichen Funktionen vorgestellt, eine größere Übersicht ist in der Dokumentation [ela16c, Input Plugins] einsehbar.

StdIn `Stdin` erlaubt das Lesen von Objekten aus der Standardeingabe. Falls nicht anders konfiguriert (z. B. mit *Multiline*-Filter), wird angenommen, dass eine Zeile ein Objekt darstellt.

File Die `File` Funktion ermöglicht es, Objekte aus Dateien auszulesen. Standardmäßig wird dabei angenommen, dass eine Zeile ein Objekt beinhaltet. Sollen mehrere Zeilen zu einem Ereignis zusammengefasst werden, müssen der *Multiline-Codec* oder ein Filter angewendet werden. Die Funktion speichert die momentane Position innerhalb der Datei in einer extra Datei mit dem Namen `sincedb`. Dies ermöglicht es, dass nach Neustart des Vorgangs nur ab der Zeile gelesen wird, die noch nicht bearbeitet wurde. Das Ändern oder Kopieren einer Datei wird von der Funktion erkannt und behandelt. Um von der neu entstandenen Datei lesen zu können, muss auch der neue Pfad zur Datei angegeben werden und wird damit als neu hinzugefügt behandelt. Damit alle Daten seit der Veränderung und dem letzten

Lesen erkannt werden, muss das Feld `start_position` auf `beginning` eingestellt sein, was dazu führt, dass sie komplett neu gelesen wird.

Http_poller Die Funktion `http_poller` ist für das Abrufen eines oder mehrerer *HTTP* Endpunkte geeignet und wandelt deren Antworten in Ereignisse um. `Urls` und `interval` sind die erforderlichen Felder bei dieser Funktion. Das Feld `interval` gibt das Intervall in Sekunden an, in welchen die *URLs* abgerufen werden sollen. Standardmäßig wird hier der *Codec* für Klartexte verwenden. Falls die Antwort aber im JSON oder anderen Formaten übermittelt wird, muss der *Codec* geändert werden. In der Logstash Dokumentation ist dafür eine vollständige Liste mit *Codecs* einsehbar [ela16c, Codec Plugins].

B. Filter

Log-Daten sind oft unstrukturiert, enthalten überflüssige Informationen oder ihnen fehlen nützliche Informationen, die man aber aus den Rohdaten noch gewinnen kann. Dafür sind viele der verfügbaren Filter ein geeignetes Mittel. Sie verschlechtern aber auch die Leistung des Systems abhängig von der Anzahl der angewendeten Filter und der Menge der zu verarbeitenden Daten.

Grok `Grok` ermöglicht es, unstrukturierte Daten in strukturierte umzuwandeln. Die Funktion basiert darauf, dass man *Pattern* passend für die Struktur der Daten erstellt um damit Objekte herauszufiltern. Standardmäßig hat Logstash mehr als 120 verschiedene *Pattern*, die online einsehbar sind [che16]. Hilfreiche Webseiten zum Konfigurieren von *Patterns* sind *Grok Debugger* [Eth16] und *Grok Constructor* [Sto16].
Die Syntax für ein Grok *Pattern* ist folgende:

```
%{PATTERN:Feldname}
```

Listing 2.10: Syntax eines Grok *Patterns*

Die Syntax besteht aus *Patterns* und den Namen des Feldes, in das der durch das *Pattern* erkannte Teil der Eingabe gespeichert wird. Standardmäßig werden alle Felder als Strings gespeichert, falls nicht anders angegeben. Dies wird durch das Anhängen eines anderen Datentypen realisiert. Momentan werden nur Umwandlungen zu Integer- und Gleitkommazahlen unterstützt.
Als kleines Beispiel für eine Eingabe dient folgende Zeile:

```
Rechnernetze 2016 Sommersemester
```

Listing 2.11: Beispieleingabe für Grok

Darauf wird der folgende Filter angewendet:

```
%{WORD:Vorlesung} %{YEAR:Jahr} %{WORD:Semester}
```

Listing 2.12: Grok Beispielpattern

Diese *Patterns* bestehen aus regulären Ausdrücken. Listing 2.13 zeigt das Resultat nach der Anwendung des Filters. Wie man sieht wurden die Daten mehr strukturiert und die Informationen der Datei wurden Feldern zugewiesen.

```
  {
  "Vorlesung": [
    [
      "Rechnernetze"
    ]
                ],
  "Jahr": [
    [
      "2016"
    ]
          ],
  "Semester": [
    [
      "Sommersemester"
    ]
              ]
}
```

Listing 2.13: Beispielresultat von Grok

Geoip `Geoip` ist eine Funktion, um zum Beispiel die Herkunft der Nutzer eines Dienstes herauszufinden. Dies wird dadurch erreicht, indem aus den gegebenen IP-Adressen die physische Adresse gewonnen wird. Logstash verwendet hier eine GeoIp-Datenbank, um die IP-Adressen in Koordinatenpaare mit der ungefähren Position, bestehend aus Längen- und Breitengrad, umzuwandeln. Logstash wird standardmäßig mit der GeoLiteCity-Datenbank geliefert, aber es können auch andere Datenbanken verwendet werden, indem unter der `database` Einstellung ein gültiger Pfad angegeben wird. Die gewonnenen Koordinaten werden in [*geoip*][*location*] Feldern mit GeoJSON-Format gespeichert. Zusätzlich werden den [*geoip*][*location*] Feldern Elasticsearch `Geo_points` zugeordnet. Diese `Geo_points` sind immer noch im GeoJSON-Format und ermöglicht somit die Verwendung von z. B. Filtern oder Anfragen.

Mutate Der `Mutate`-Filter erlaubt es, Felder auf verschiedene Weisen zu modifizieren. Unter anderem kann man Felder umbenennen, entfernen oder ersetzen.
Beispiel:

```
 filter {
  mutate {
    # Das Feld "Beispiel" wird hier umbenannt zu "Beispiel123"
    rename => { "Beispiel" => "Beispiel123" }
  }
}
```

Listing 2.14: Umbenennung eines Feldes

C. Output

Output ist die letzte Station für die Daten in Logstash. Hier werden die Daten an ihren Zielort geschickt. Dies wird wieder durch Funktionen realisiert, von denen ein paar nun vorgestellt werden. Eine größere Übersicht ist in der Dokumentation [ela16c, Output Plugins] einsehbar.

Elasticsearch Die Elasticsearch Funktion wird genutzt, um die Daten an Elasticsearch zu senden, wo sie dann gespeichert werden können. Im Folgenden wird nur ein kleiner Teil der verfügbaren Optionen vorgestellt.

Index Die ausgehenden Daten müssen durch Angabe eines Stringfeldes indexiert werden, um sie einem Elasticsearch Index zuzuweisen.
Beispiel:

```
output {
  Elasticsearch {
    # Der Index speichert hier die Woche der Indexierung
    index => "Logstash"-%{+xxxx.ww}"
  }
}
```

Listing 2.15: Definition eines Zielindexes

Hier wird als Index der Tag der Einspeisung in das System gespeichert, was das Suchen nach Altdaten erleichtert.

Hosts In diesem Feld werden die Adressen in Listen gespeichert, an die Logstash den *Output* weiterleitet. Falls nicht anders definiert, ist hier der Wert `127.0.0.1` der Standard.

document_id Die `document_id` kann dazu verwendet werden, um Dokumente eines Indexes zu identifizieren. Somit können Duplikate erkannt werden oder bereits bestehende Dokumente durch neue mit der selben Identifikationsnummer überschrieben werden.
Beispiel:

```
output {
  Elasticsearch {
   document_id => "%{Name des Feldes}"
  }
}
```

Listing 2.16: Konfiguration einer `document_id`

Nagios Die Nagios Funktion unterstützt Nagios 3, um über die Nagios Kommandozeile passive *Checkresults* an Nagios zu senden. Dazu werden die Felder `nagios_host` und `nagios_service` benötigt, während `nagios_annotation` und `nagios_level` nur optional sind. Des Weiteren kann der Pfad zu dem externen *Nagios Command File* angegeben werden, falls dieser nicht dem Standard entspricht, wie auch das *Level* der zu sendenden *Checks*.

Email Um die durch Logstash erstellte Ausgabe per Email zu verschicken ist diese Funktion geeignet. Das einzig benötigte Feld ist das String Feld `to`, bei dem die Emailadressen der Empfänger angegeben werden müssen.

2.1.2. Elasticsearch

In diesem Kapitel wird es lediglich um die grundlegende Benutzung von Elasticsearch und den damit verbundenen Möglichkeiten gehen, die für die Benutzung des *ELK-Stacks* relevant

sind, da dies sonst den Rahmen dieser Arbeit sprengen würde. Das Wissen aus diesem Kapitel basiert auf dem Handbuch von Clinton Gormley und Zachary Tongs [CG15].

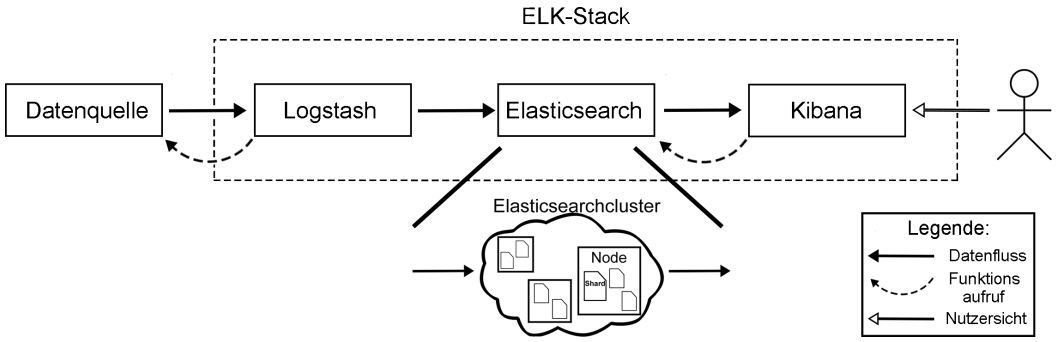

Abbildung 2.3.: Rolle von Elasticsearch im *ELK-Stack*

Allgemeines

Elasticsearch basiert auf der Programmbibliothek zur Volltextsuche Lucene. Das *Open-Source* Projekt der Firma Elastic ermöglicht unter anderem die verteilte Speicherung, skalierbare Volltextsuche in Echtzeit und Analyse von Daten. Antworten auf Anfragen werden im JSON-Format ausgegeben. Elasticsearch ist dokumentenorientiert, d. h. es werden Dokumente in Indexen gespeichert. Diese Dokumente beinhalten Daten im JSON-Format, der *JavaScript Object Notation*. Mit Elasticsearch kann entweder über Java (Port 9300) oder mit der *RESTful API* (Port 9200) kommuniziert werden. Einstellungen wie Port oder Hostadresse eines Knoten können unter `/etc/elasticsearch/elasticsearch.yml` konfiguriert werden. Verwendet wird Elasticsearch unter anderem von Amazon Web Services [Ama16] und Wikimedia [Hor16]. Wie in Abbildung 2.3 zu sehen ist, hat Elasticsearch die Rolle der Speicherung und Erstellung von Antworten auf Anfragen durch Kibana.

Cluster, Nodes und Shards

Elasticsearch ist so konzipiert, einfach und ohne viel Aufwand zu skalieren. Obwohl die vertikale Skalierung möglich ist, d. h. das Verwenden von Rechnern mit mehr Leistung, ist die horizontale Skalierung, bei der die Anzahl der Rechner erhöht wird, effizienter und zuverlässiger, da z. B. der Datenverlust bei einem Medienfehler durch Sicherungskopien auf einem der anderen Rechner verhindert werden kann. Jedes Elasticsearch System besteht aus einem Verbund von einem oder mehreren Rechnern, auch *Cluster* genannt. Jeder dieser Rechner stellt einen Knoten dar, auch *Nodes* genannt, die sich untereinander die Daten- und Rechenlast aufteilen, wobei ein Knoten im Rechnerverbund immer der Hauptknoten sein muss. Hauptknoten sind dafür verantwortlich, Veränderung im Rechnerverbund wie das Hinzufügen eines Index oder das Entfernen eines Knoten zu verwalten. Nutzer können Anfragen an jeden beliebigen Knoten im Verbund stellen, da jeder Knoten den Ort eines gespeicherten Dokuments kennt. Die Anfrage kann somit an die zuständige Stelle weiterleiten werden, um dem Nutzer die Antwort zu liefern. Knoten wiederum bestehen aus Scherben (englisch *Shards*), die die Dokumente indexiert enthalten und verteilt über die Knoten gespeichert werden. Jede Scherbe stellt dabei eine selbstständige Lucene Instanz dar und somit eine vollwertige Suchmaschine. Es gibt 2 verschiedene Arten von Scherben, zum einen die Primärscherben (englisch *primary*

shards), die eine, von der Hardware abhängige, begrenzte Anzahl (bis zu 128) von Dokumenten beinhalten. Jedes Dokument ist dabei nur in einer einzigen Primärscherbe gespeichert. Replikascherben (englisch *replica Shards*) stellen Sicherungskopien von Primärscherben dar, um bei Medienfehlern oder bei sehr vielen Such- bzw. Leseanfragen eine Alternative zu einer Primärscherbe zu ermöglichen. Die Anzahl der Primärscherben kann vor dem Erstellen eines Indexes konfiguriert werden, danach aber nicht mehr. Bei Replikascherben hingegen lässt sich deren Anzahl pro Primärscherben auch noch nach Erstellen eines Indexes verändern. Falls es mehr als einen Knoten im Rechnerverbund gibt, werden die Primär- und Replikascherben wenn möglich so aufgeteilt, dass bei einem Ausfallen eines Knoten trotzdem noch alle Dokumente verfügbar sind. Neue Dokumente werden erst auf der Primär und dann parallel auf allen zugehörigen Replikascherben gespeichert. Dieser Mechanismus wird in den folgenden Paragraphen an einem Beispiel demonstriert.

Beispielausprägungen eines Clusters In dem folgenden Beispiel werden einem Rechnerverbund zusätzliche Knoten und Scherben hinzugefügt. Zu Beginn besitzt der Verbund 2 Knoten und 3 Primärscherben mit je einer Replikascherbe. Somit könnte schon jetzt ein Ausfall eines Knoten verkraftet werden, da jeder von ihnen alle Dokumente besitzt.

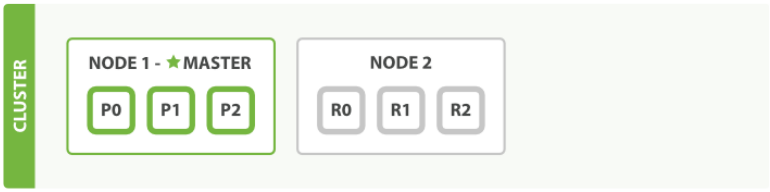

Abbildung 2.4.: Beispiel mit 2 Knoten, 3 Primärscherben mit je einer Replikascherbe[CG15, add_failover]

Wird nun noch ein Knoten eingefügt, werden die Primärscherben automatisch so verteilt, sodass die Ausfallsicherheit und die Performance steigt, da sich nun weniger Scherben einen Knoten teilen müssen.

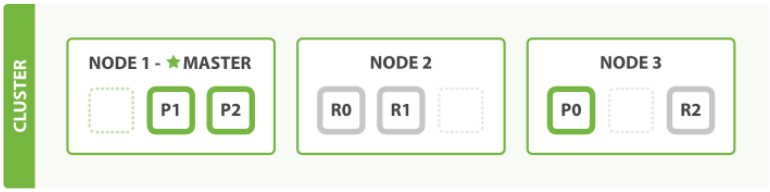

Abbildung 2.5.: Beispiel mit 3 Knoten, 3 Primärscherben mit je einer Replikascherben[CG15, scale_horizontally]

Wird nun die Zahl der Replikascherben pro Primärscherbe auf zwei erhöht, sieht der Rechnerverbund folgendermaßen aus:

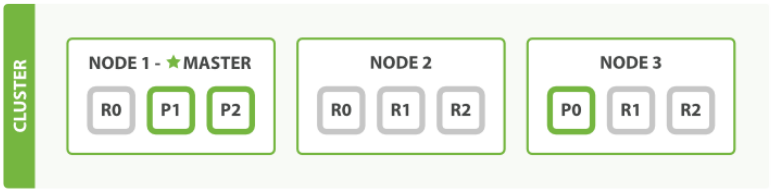

Abbildung 2.6.: Beispiel mit 3 Knoten, 3 Primärscherben mit je zwei Replikascherben[CG15, scale_horizontally]

Mit dieser Konfiguration wäre der Ausfall von 2 Knoten immer noch verkraftbar, da auf jedem Knoten die Daten aller Scherben gespeichert werden. Falls einer der ausgefallenen Knoten der Hauptknoten war, müssen sich die übrigen sofort auf einen neuen Hauptknoten einigen.

Cluster Health Der Status eines Rechnerverbunds kann drei verschiedene Ausprägungen besitzen:
Grün: Alle Primär- und Replikascherben sind aktiv.
Gelb: Alle Primär-, aber nicht alle Replikascherben sind aktiv.
Rot: Nicht alle Primärscherben sind aktiv.
Ein Beispiel für einen gelben Status wäre zum Beispiel ein Rechnerverbund mit einem Knoten und einer Replikascherbe pro Primärscherbe. Dann wären keiner der Replikascherben aktiv, da es keinen Sinn ergibt, diese auf dem selbem Knoten wie einer Primärscherbe zu speichern, weil dies beim Ausfall des Knoten keine Ausfallsicherung garantieren würde.

Suche

In Elasticsearch gibt es grundsätzlich zwei verschiedene Arten eine Suchanfrage zu stellen. Zum einen gibt es die *Lite*-Suche, bei der die ganze Suchanfrage mit ihren Parametern aus einem Anfragestring besteht. Zum anderen gibt es die Suche mithilfe der *Query Domain Specific Language* (kurz *DSL*), bei der die Anfrage aus einem JSON Objekt besteht. Die *Lite*-Suche ist kompakter und bei wenigen Parametern schneller, aber bei komplexeren Suchen sehr unübersichtlich und daher auch schwer zu debuggen. Standardmäßig werden aus Performancegründen, falls nicht anders definiert, nur die Top 10 der Treffer zurückgegeben. Jede Suchanfrage ist folgendermaßen aufgebaut und kann z. B. mithilfe des Kommandozeilen-Programms cURL ausgeführt werden. Tabelle 2.1 beschreibt die Bedeutung der jeweiligen Felder.

```
<VERB> '<PROTOCOL>://<HOST>:<PORT>/<PATH>?<QUERY_STRING>'
                    -d '<BODY>'
```

Listing 2.17: Aufbau einer Anfrage an Elasticsearch

In Listing 2.18 sind einige Beispiele für solche Anfragen zu sehen. Die Anfrage in Zeile 1 liefert die Dokumente aller Indexe, die mit dem Namen mit `index-` beginnen, wohingegen Zeile 2 alle Dokumente innerhalb des gesamten Rechnerverbundes zurückgibt. Im Folgenden wird ein gekürztes Format verwendet, dass man in der Sense Konsole benutzen kann. Sense ist eine Anwendung innerhalb von Kibana, die es ermöglicht, Anfragen direkt in den Browser

einzugeben. So wird aus der Anfrage in Zeile 2 die Anfrage aus Zeile 3. Wie in Zeile 4 zu sehen ist, können auch mehrere bestimmte Indices durchsucht werden (In diesem Fall die Indexes `index1` und `index2`).

```
GET 'http://localhost:9200/index-*/_search'
GET 'http://localhost:9200/_search' '{}'
GET /_search {}
GET /index1,index2/_search
```

Listing 2.18: Beispiele für Anfragen an Elasticsearch

`VERB`	Dieses Feld legt die Anfragemethode vor. z. B. `GET` oder `DELETE`
`PROTOCOL`	Verwendetes Netzwerkprotokoll. Entweder `http` oder `https`
`HOST`	Hostname des Knoten eines Elasticsearch Rechnerverbundes (z. B. `localhost`).
`PORT`	Der Port, auf dem Elasticsearch läuft. (Normalerweise 9200)
`PATH`	Gibt den Endpunkt der *API* an, der mehrere Komponenten beinhalten kann.
`QUERY_STRING`	Optionales Feld für Parameter. z. B. `?pretty` zur leserlichen Ausgabe.
`BODY`	Optionales Feld zur Anfrage im JSON Format.

Tabelle 2.1.: Felder einer Anfrage in Elasticsearch

Lite Die *Query* String Suche, bei der die komplette Suchanfrage in einem String übergeben wird, ist sinnvoll, wenn es sich um kurze Anfragen handelt, wie die Ausgabe aller Dokumente innerhalb eines Rechnerverbunds oder Index. Zeichen, die bei der *URL*-Kodierung reserviert sind, müssen hier durch die Prozentdarstellung maskiert werden, was die Suchanfrage für den Benutzer unverständlicher macht.
Im folgenden Beispiel werden alle Datensätze über Spiele ausgegeben, die gewonnen wurde und bei denen der Held `Akali` ausgewählt wurde:

```
GET index/_search?q=%2Bchampionname%3AAkali+%2Bgames.stats.win%3Atrue
```

Listing 2.19: Beispiel für eine Lite Anfrage

Wie man an dieser relativ kurzen Anfrage schon erkennen kann, werden die *URLs* schnell unübersichtlich und auch schwerer zu debuggen.

Query DSL Um komplexere Suchanfrage auszuführen ist die *Query DSL* geeigneter, weil das JSON Format mehr Übersichtlichkeit und Funktionalität bietet. Die Anfragen werden aus Anfrageklauseln mit dem folgenden Schema zusammengesetzt[CG15, Query DSL]:

```
{
    QUERY_NAME: {
        ARGUMENT: VALUE,
        ARGUMENT: VALUE,
                }
}
```

Listing 2.20: Aufbau einer Anfrageklausel

Es gibt dabei zwei Arten von Klauseln, Blatt- und Verbindungsklauseln. Blattklauseln (englisch *leaf queries*) werden benutzt, um eins oder mehrere Felder mit einem *query string*

zu vergleichen. Verbindungsklauseln (englisch *compound queries*) hingegen ermöglichen die Kombination von Klauseln. Beispiel für eine Blattklausel mit einer `match` Bedingung ist das folgende Listing, in dem nach Dokumenten gesucht wird, in denen das Feld `championname` den String `Akali` beinhaltet.

```
GET /_search
{
    "query": {
        "match": {
            "championname": "Akali"
        }
 }}
```

Listing 2.21: Anfrage mit einer Bedingung

Als Beispiel für eine Verbindungsklausel wird die `bool` Bedingung verwendet, die drei `match` Bedingungen kombiniert, die entweder vorhanden, nicht vorhanden oder, wenn möglich, vorhanden sein müssen:

```
GET logstashrecentsloron/_search/
{
"query": {
    "bool": {
        "must":     { "match": { "championname": "Akali" }},
        "must_not": { "match": { "games.stats.win": false }},
        "should":   { "match": { "games.gameMode": "CLASSIC" }}
            }
}}
```

Listing 2.22: Anfrage mit mehreren Bedingungen

Das Verhalten einer Klausel hängt auch davon ab, in welchen Kontext sie eingebunden ist. Falls sie wie im obigen Beispiel in einer Anfrage eingebettet ist, wird ein Wert berechnet, inwieweit ein Dokument der Suche entspricht. Die Suchergebnisse mit den höchsten Werten werden als Treffer angezeigt. Der andere Kontext ist der des Filters. Hierbei werden keine Werte errechnet, sondern lediglich festgestellt, ob ein Dokument die Bedingungen zu 100 Prozent erfüllt oder nicht.

Aggregationen

Aggregationen in Elasticsearch werden, wie eine Suchanfrage, ebenfalls durch JSON Dokumente realisiert und können miteinander kombiniert werden. So kann zum Beispiel zu einer Suche noch ein Durchschnittswert eines bestimmten Feldes abgefragt werden oder die Häufigkeit eines Wertes. Aggregationen bestehen aus Kombinationen von *Buckets* und *Metrics*. *Buckets* sind hierbei eine Menge von Dokumenten, die eines oder mehrere Kriterien erfüllen. *Metrics* sind Werte, die aus den Dokumenten eines *Buckets* errechnet werden, wie der Durchschnittspunktezahl einer Liga. Ein *Bucket* kann zum Beispiel alle Dokumente von Spielen enthalten, die gewonnen oder verloren wurden, wie im folgenden Beispiel zu sehen ist.

```
GET logstashrecentsloron/_search
{
    "size" : 0,
    "aggs" : {
```

```
        "games_won" : {
            "terms" : {
              "field" : "games.stats.win"
                       }
}}}
```

Listing 2.23: Durchführen einer Aggregation

Die Dokumente des Index `logstashrecentsloron` werden hier in die *Buckets* unterteilt, die den gleichen Wert im Feld `games.stats.win` besitzen, also in diesem Fall, `true` oder `false`. Der Name der Aggregation ist frei wählbar, in diesem Fall ist er `games_won`. Da `size` auf null gesetzt ist, werden keine Suchergebnisse angezeigt, sondern nur die Aggregationsergebnisse:

```
{
[...]
 "hits": {
    "total": 10,
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "games_won": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": 0,
          "key_as_string": "false",
          "doc_count": 5
        },
        {
          "key": 1,
          "key_as_string": "true",
          "doc_count": 5
        }
                  ]
}}}
```

Listing 2.24: Ergebnisse einer Aggregation

`Doc_count` ist die Anzahl der Dokumente, die dem *Bucket* zugeordnet wurden. Um nun eine *Metric* hinzuzufügen, wird eine weitere Aggregation hinzugefügt mit dem Namen `avg_kills`, die die durchschnittliche Anzahl der Tötungen pro Spiel für ein *Bucket* ausgeben soll. Dafür wird eine *nested Metric* benötigt, also muss die neue Aggregation mit der *Metric* `avg` in die bereits bestehende integriert werden. Als Feld für die Berechnung wird hier `games.stats.championsKilled` benutzt, da es die Anzahl der Tötungen des jeweiligen Spiels enthält.

```
GET logstashrecentsloron/_search
{
    "size" : 0,
    "aggs" : {
        "games_won" : {
            "terms" : {
              "field" : "games.stats.win"
            },
```

```
        "aggs":    {
            "avg_kills": {
                "avg": {
                    "field": "games.stats.championsKilled"
                        }
}}}}}
```

Listing 2.25: Verwendungen mehrerer Aggregationen

Diese Anfrage liefert folgende Antwort:

```
{
[...]
"hits": {
    "total": 10,
    "max_score": 0,
    "hits": []
  },
  "aggregations": {
    "games_won": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": 0,
          "key_as_string": "false",
          "doc_count": 5,
          "avg_kills": {
            "value": 8.6
          }
        },
        {
          "key": 1,
          "key_as_string": "true",
          "doc_count": 5,
          "avg_kills": {
            "value": 10}
                ]
}}}}
```

Listing 2.26: Ergebnis der Abfrage mittels mehrerer Aggregationen

Mit solchen Anfragen lassen sich Schlüsse über Zusammenhänge ziehen, wie man an Listing 2.26 sieht. Hier ist die durchschnittliche Tötungsrate eines Spielers höher bei gewonnenen Spielen als bei verlorenen. Der Durchschnitt ist dabei nur eine von viele vorhandenen *Metrics*. Es gibt unter anderem auch noch `min`, `max` oder `sum`. Es ist auch möglich *Buckets* in anderen *Buckets* zu verschachteln, wie im Folgenden:

```
GET logstashrecentsloron/_search
{
    "size" : 0,
    "aggs" : {
        "games_won" : {
            "terms" : {
              "field" : "games.stats.win"
            },
        "aggs": {
```

```
            "avg_kills": {
               "avg": {
                  "field": "games.stats.championsKilled"
               }
            },
         "champs": {
               "terms": {
                  "field": "champion" }
                  }
}}}}
```

Listing 2.27: Verschachtelung von *Buckets*

In diesem Beispiel wird ein *Bucket* hinzugefügt, der die Identifikationsnummern der gespielten Charaktere in den verlorenen bzw. gewonnenen Spielen enthält. Damit lässt sich ein Rückschluss darüber ziehen, ob ein Charakter öfter gewinnt oder nicht. So lassen sich Aggregation beliebig ineinander verschachteln.

2.1.3. Kibana

In diesem Kapitel werden die verschiedenen Funktionen von Kibana, die zum Teil aufeinander aufbauen, behandelt. Dieses Kapitel basiert auf dem *Kibana User Guide* [ela16b].

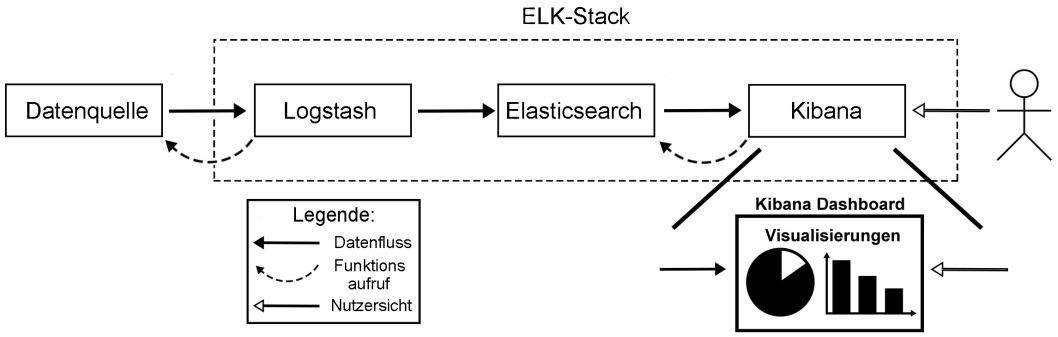

Abbildung 2.7.: Einordnung von Kibana im *ELK-Stack*

Allgemeines

Kibana ist seit Version 4 eine NodeJS-Anwendung mit einem AngularJS Front-End zum Visualisieren der durch Elasticsearch bereitgestellten Daten. So können Nutzer zum Beispiel *Dashboards* aus Visualisierungen und Kennzahlen mit den Daten eines Index erstellen. Kibana bekommt die Daten von Elasticsearch nach einen Funktionsaufruf geliefert, wie man in Abbildung 2.7 sieht. So werden die Funktionen von Elasticsearch zum Beispiel zur Abfrage von Aggregationen genutzt, um damit Histogramme erstellen zu können. Servereinstellungen wie Port oder IP-Adresse können unter `/opt/kibana/config/kibana.yml` konfiguriert werden.

Discover

Unter dem *Discover* Reiter (siehe Abbildung 2.8) ist jedes Dokument des aktuell ausgewählten Indexes einsehbar. Es ist möglich, die Daten zu durchsuchen, zu filtern und zu betrachten.

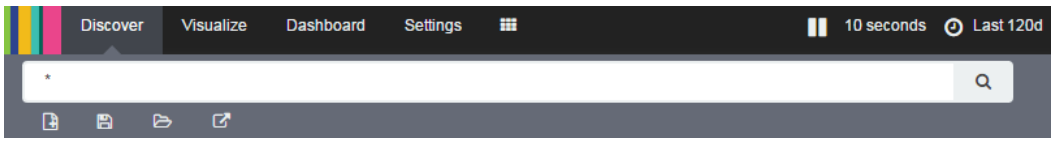

Abbildung 2.8.: Übersicht der Funktionen in Kibana

Es werden auch Statistiken zu der eingegeben Suche bereitgestellt, wie die Anzahl der Treffer. Falls der Index einen Zeitstempel besitzt, ist auch die Darstellung der Verteilung der Suchergebnisse über ein zeitbasiertes Histogramm möglich.

Suchen Die Suchleiste (siehe Abbildung 2.8) versteht unterschiedliche Eingabesyntaxe. Die einfachste Option ist die simple Eingabe von Strings ohne bestimmte Syntax. Aber es sind auch die Eingaben nach der *Lucene query* Syntax oder der *Elasticsearch Query DSL* zulässig. Die folgenden Abschnitte handeln von der *Elasticsearch Query DSL* für die Volltextsuche und deren Syntax. Allgemeine Beispiele für Suchanfragen sind in Tabelle 2.2 zu sehen. Es ist möglich eine Suche unter einem bestimmten Namen zu speichern und wieder zu laden. Auch ist der zu durchsuchende Index veränderbar.

Suche:	Ergebnis:
`games.stats.win:false`	Gibt alle Suchergebnisse wieder, bei denen das Feld `games.stats.win` `false` ist.
`games.stats.win:(false OR true)`	Gibt alle Suchergebnisse wieder, bei denen das Feld `games.stats.win` `false` oder `true` ist.
`_exists_:champions.stats.totalAssists`	Gibt alle Suchergebnisse wieder, bei denen das Feld nicht null ist.
`_missing_:champions.stats.totalAssists`	Gibt alle Suchergebnisse wieder, bei denen das Feld null oder nicht vorhanden ist.
`player: "Max Mustermann"`	Gibt alle Suchergebnisse wieder, bei denen das Feld `player` genau den String `Max Mustermann` enthält.
`champions.stats.\*:(100 50)`	Liefert alle Suchergebnisse, bei denen die Feldernamen mit `champions.stats` beginnen und 50 oder 100 enthalten. (z. B. `champions.stats.totalAssists`)

Tabelle 2.2.: Beispiele für Suchanfragen

In den folgenden Paragraphen werden Möglichkeiten zur gezielten Suche behandelt, die in der Suchleiste verwendet werden können.

Wildcards Wildcards können in der Suche verwendet für einzelne Zeichen (?) werden, aber auch für null oder mehr (*).
Beispiel:

```
Suche: player:"Ma? Mustermann"
Suche: player:"Max Muster*"
```

Listing 2.28: Beispiele für Wildcards

Bei der Verwendung muss darauf geachtet werden, dass Wildcards enorme Speicherauslastung bewirken können und sich so negativ auf die Performance auswirken. Beispielsweise müssen bei der Suchanfrage `*ing` alle Dokumente nach Wörtern durchsucht werden, die auf `ing` enden, was einen erheblichen Aufwand bedeuten kann. Darum wird die Analyse von Treffern bei Wildcards standardmäßig deaktiviert, da Wörter mit fehlenden Buchstaben nur sehr schwer zu analysieren sind.

Reguläre Ausdrücke Auch reguläre Ausdrücke können verwendet werden, indem sie mit Vorwärtsschrägstrichen umhüllt werden. Die Syntax der regulären Ausdrücke kann in der Dokumentation [CG15, Regexp Query] eingesehen werden.
Beispiel:

```
Suche: player:/Ma. (Mustermann)/
```

Listing 2.29: Beispiele für reguläre Ausdrücke

Fuzziness Es ist möglich nach Wörter zu suchen, die dem gesuchten Wort ähneln, aber nicht identisch mit ihm sind. Dies wird durch das Tilde Zeichen (~) realisiert. Dabei werden alle Wörter gesucht, die sich höchstens um zwei Zeichen vom gesuchten unterscheiden, was auf der Levenshtein-Distanz basiert. Die Zeichen können dabei gelöscht, eingefügt, einzelne ausgetauscht oder zwei benachbarte vertauscht werden. Man kann den Zeichenunterschied auch auf eins begrenzen, indem nach der Tilde `1` steht.
Beispiel:

```
Suche: Mxa~
Suche: Mal~1
```

Listing 2.30: Beispiele für *Fuzziness*

Ähnlichkeitssuche Falls die gesuchten Wörter auch in einer anderen als der gegeben Reihenfolge oder auch mit anderen Wörter dazwischen vorkommen können, kann ebenfalls ein Tildezeichen verwendet werden mit einer maximal Distanz der Wörter. Dabei werden die Ergebnisse mit der Distanz aufsteigend geordnet.
Beispiel:

```
Suche : "Max Mustermann"~3
```

Listing 2.31: Beispiel für Ähnlichkeitssuche

Intervalle Intervalle können für Zahlen, Strings und Daten definiert werden. Dabei werden offene Intervalle mit geschweiften Klammern ({min TO max}) geschrieben und abgeschlossene mit eckigen ([min TO max]). Beide Typen sind auch miteinander kombinierbar.
Beispiel:

```
Suche: Datum:[2012-01-01 TO 2015-01-01]
Suche: Datum:{* TO 2015-12-31]
Suche: Alter: [1 TO 18}
Suche: Alter:(>=18 AND <30)
```

Listing 2.32: Beispiele für Intervalle

Priorisierung Durch Benutzen des ^ Zeichens ist es möglich nach bestimmten Begriffen priorisiert zu suchen. Dabei ist der Standardwert eins, kann aber auch jede andere positive Gleitkommazahl betragen. Werte zwischen null und eins bewirken, dass Begriffe an Prioriät verlieren. Es können auch Gruppen und Phrasen priorisiert werden.
Beispiel:

```
Suche: Max^2 Mustermann^0.5
Suche: "Max Musterman"^3
Suche: (Max Mustermann)^4
```

Listing 2.33: Beispiele zur Priorisierung

Optionale Begriffe Falls die Standardeinstellungen nicht verändert wurden, sind alle Begriffe optional, solange einer gefunden wird. Mit den Operatoren + und - ist es aber möglich zu definieren, dass bestimmte Begriffe vorhanden bzw. nicht vorhanden sein müssen. Dabei müssen Begriffe mit dem + Operator vorhanden sein, Begriffe mit dem - Zeichen hingegen sind optional.
Beispiel:

```
Suche : +Max -Mustermann
```

Listing 2.34: Beispiel für optionale Begriffe

Es können auch die Operatoren AND/&& , OR/| | und NOT/! verwendet werden. Dabei muss darauf geachtet werden, dass bei den +/- Operatoren jeweils nur die rechten Begriffe betroffen sind, bei AND/OR aber die linken und rechten Begriffe, was die Komplexität erhöhen kann.

Gruppierungen Durch die Umklammerung mit runden Klammern () können Gruppen gebildet werden. Gruppen können bei Feldern, aber auch bei Priorisierungen verwendet werden.
Beispiel:

```
Suche: (Max Mustermann)^3
Suche: games.stats.win:(false OR true)
```

Listing 2.35: Beispiel für Gruppierungen

Reservierte Zeichen Folgende Zeichen sind reserviert und darum mit einem *Backslash* (\) zu maskieren:

```
+ − = && || > < ! ( ) { } [ ] ^ ” ~ * ? : \ /
```

Listing 2.36: Reservierte Zeichen

Aktualisieren Suchanfragen, wenn gewollt, können je nach eingestellten Intervall automatisch wiederholt und auch auf Zeiträume begrenzt werden. Das automatische Aktualisieren kann pausiert werden.

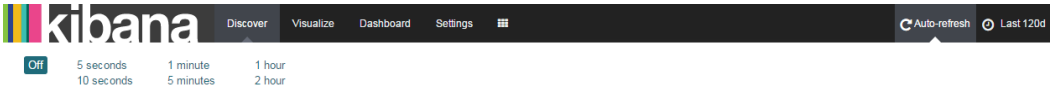

Abbildung 2.9.: Aktualisieren von Daten in Kibana

Felderbasiertes Filtern Suchergebnisse können gefiltert werden, indem nur Felder mit einem bestimmten Wert angezeigt oder nicht angezeigt werden. Neu hinzugefügte Filter, die durch die Felderliste oder von der Dokumententabelle hingezugefügt werden können, werden in der Filterleiste unter der Suchanfrage angezeigt. Filter können dort aktivert bzw. deaktiviert, gelöscht oder auch invertiert (z. B. Datensätze mit bestimmten Feldwerten werden nun ausgeschlossen werden). Es ist möglich, benutzerdefinierte Filter mit JSON Format unter *Custom Filter* zu erstellen.

Abbildung 2.10.: Feldbasierters Filtern in Kibana

Visualize

Unter dem *Visualize* Reiter (siehe Abbildung 2.8) lassen sich Visualisierungen erstellen, unter einem Namen speichern und auch bereits gespeicherte wieder laden. Diese können auch später im *Dashboard* geladen werden. Es gibt grundsätzlich 8 verschiedene Typen von Visualisierungen, die nach gewählten Index mit dessen Daten gespeist werden und sich auch wieder automatisch, wenn gewollt, nach einer bestimmten Zeit aktualisieren. Für den Versuchsaufbau sind manche davon jedoch hinfällig, da zum Beispiel für die *Tile Map* die benötigten geographischen Daten fehlen. Auch hier kann wieder wie schon im *Discover* Reiter feldbasiert gefiltert werden.

Grafiken Folgende Visualisierungsmöglichkeiten stehen in Kibana zur Verfügung:

- *Vertical bar chart* (Säulendiagramm)
- *Pie chart* (Kreisdiagramm)
- *Line chart* (Liniendiagramm)
- *Area chart* (Flächendiagramm)

Data table Mithilfe von *data tables* können Aggregationen in Form von Tabellen dargestellt werden.

Metric Diese Visualisierung besteht aus Kennzahlen, die mithilfe von verschiedenen Aggregationen wie Durchschnitt, Maximum oder Summe errechnet werden.

Markdown Widget Das *Markdown Widget* ist ein Textfeld, dass die *GitHub Flavored Markdown Syntax* akzeptiert. Hier können zum Beispiel Texte kursiv, als Überschrift oder als Liste formatiert werden.

Dashboard

Ein *Dashboard* lässt sich aus den gespeicherten Visualisierungen zusammensetzten. Die Größe und Position von Visualisierungen innerhalb des *Dashboards* lassen sich beliebig verändern. Die für das *Dashboard* verwendeten Daten können zeitlich begrenzt werden und automatisch in bestimmten Abständen aktualisiert werden. *Dashboards* können unter einem bestimmten Namen gespeichert bzw. gespeicherte geladen werden. Die Daten, die für die Visualisierungen eines *Dashboards* verwendet werden, können durch die Eingabe einer Suche begrenzt werden.

Plugins

Kibana unterstützt die Verwendung von *Plugins* seit der Version 4.2.0 in Form von z. B. zusätzlichen Erweiterungen oder Visualisierungen. Diese Erweiterungen können mithilfe des `plugin` Befehl installiert und verwaltet werden.

Bestehende Plugins Neben dem schon erwähnten Sense, das zur bequemeren Eingabe von Anfragen an Elasticsearch dient, gibt es eine ganze Reihe von inoffiziellen und offiziellen *Plugins* mit verschiedenen Verwendungsmöglichkeiten.

Shield Das *Shield Plugin* ermöglicht den Schutz vor unerlaubten Zugriffen auf Kibana. Dies wird unter anderem realisiert durch rollenbasierte Zugriffskontrolle, Authentifizierung oder *SSL/TLS* Verschlüsselung.

Kibana Heatmap Plugin Mit diesem *Plugin* ist es möglich, *Heatmaps* zu erstellen und auch einem *Dashboard* hinzuzufügen.

Kibana Radar Chart Plugin Das *Radar Chart Plugin* ermöglicht die Erstellung von Netzdiagrammen mit 3 oder mehr Achsen.

Entwicklung von Plugins Für die Entwicklung von eigenen *Plugins* ist es sinnvoll, die Entwicklerversion von Kibana, bei der Funktionen wie *Debugging* oder Testen möglich sind, npm und NodeJS zu installieren. Die Kompatibilität von *Plugins* mit kommenden Versionen von Kibana ist nicht gewährleistet. Jedes *Plugin* ist ein npm Module und benötigt mindestens die Dateien `package.json` und `index.js`. In `package.json` werden Informationen wie der Name oder die Version der Erweiterung gespeichert. `Index.js` beinhaltet das npm Modul und den gewünschten *Code*. Alles, was später einem Nutzer über das *Front-End* ausgegeben

werden soll, wird in einem Unterverzeichnis namens `public` gespeichert. Es besteht auch die Möglichkeit der Verwendung eines *Kibana Plugin Yeoman Generators* der bei der Erstellung der benötigten Dateien hilft.

2.2. League of Legends und Riot API

Dieser Abschnitt liefert einen kurzen Überblick über das Spiel *League of Legends* und die zugehörige *API*, die verwendet wird, um die Spieldaten abzurufen. Dabei wird darauf eingegangen, welche Daten anfallen, wie sie abgerufen werden können, in welcher Form sie vorliegen und warum sie als Anwendungsbeispiel für den *ELK-Stack* geeignet sind.

2.2.1. League of Legends

League of Legends ist ein *Online Multiplayer* Spiel, das 2009 von Riot veröffentlicht wurde und dem Genre *MOBA* (*Multiplayer Online Battle Arena*) angehört. Es gibt mehrere über die Welt verteilte Server, um unter anderen eine bessere Latenz zu ermöglichen. So gibt es zum Beispiel einen Server für Westeuropa (EUW) und einen für Nordamerika (NA).

Spielprinzip Um zu verstehen, welche Kennzahlen und Statistiken von Bedeutung sind, ist es unerlässlich, das Spielprinzip von *League of Legends* zu erfassen. Darum wird im Folgenden eine kurze Zusammenfassung gegeben. Hier wird der am meisten gespielte und für die professionelle Szene relevante Spielmodus, das 5 gegen 5 (auch Kluft der Beschwörer genannt), behandelt. Dabei treffen 2 Mannschaften mit je 5 Spielern aufeinander.

Ziel eines Spieles Das Ziel jedes Spieles ist es, das gegnerische Hauptgebäude, auch Nexus genannt, zu zerstören. Der Nexus wird aber von zahlreichen Türmen, Inhibitoren, Vasallen (siehe Abbildung 2.12) und von 5 Spielern, von denen jeder einen Helden (auch Charakter oder *Champions* genannt) kontrolliert, beschützt. Das Spielfeld ist so aufgeteilt, dass jedes der zwei Teams eine Hälfte besitzt. Bevor der Nexus zerstört werden kann, müssen daher diese Hindernisse so gut es geht beseitigt werden.

Helden Helden sind Einheiten, die von einem einzelnen Spieler gesteuert werden. Momentan gibt es 134 Helden [Rio17a]. Jeder Held hat eine einzigartige Kombination von Fähigkeiten und Stärken. Fähigkeiten erhöhen z. B. das Lauftempo oder fügen einer gegnerischen Einheit Schaden zu. Wird einem Helden z. B. 200 Schaden zugefügt, verliert er 200 Lebenspunkte. Wenn die Lebenspunkte eines Heldes unter 0 fallen, stirbt er und wird erst nach einer bestimmten Zeit wiederbelebt. Während dieser Zeit kann er nicht ins Spielgeschehen eingreifen. Umso weniger Helden einer Mannschaft am Leben sind, umso weniger kann der Nexus beschützt werden. Darum versuchen sich die Mannschaften gegenseitig zu töten, damit das Ziel des Spieles (die Zerstörung des Nexus) erreicht werden kann. Wenn ein Held einen gegnerischen Helden Schaden zufügt, der ihn unter 0 Lebenspunkte fallen lässt, bekommt er die Tötung zugeschrieben. Tötungen gewähren Erfahrung und Gold, die zur Verbesserung der Attribute eines Helden benötigt wird. Hilft ein Held bei der Tötung, aber führt nicht den letzten Schlag aus, bekommt er eine Unterstützung zugeschrieben, die auch mit Gold und Erfahrung belohnt wird, aber weniger als bei einer Tötung.

Attribute von Helden Die Attribute und Fähigkeiten eines Helden werden verbessert durch Stufenaufstiege, Runen, Gegenstände und Meisterschaften. Runen, Meisterschaften und der Held selbst müssen vor dem Start des Spiels festgelegt werden. Währenddessen werden Gegenstände und Stufen der Charaktere während des Spiels aufgebessert (siehe Abbildung 2.11). Gegenstände können durch Gold gekauft werden, welches man genauso wie Erfahrung erhält, indem man zum Beispiel gegnerische Vasallen oder Helden zerstört. Jeder Held hat bestimmte Fähigkeiten und Stärken, die durch das Auswählen der richtigen Gegenstände, Runen und Meisterschaften noch gesteigert werden können. Zum Beispiel gibt es Helden, die wesentlich mehr als andere von Gegenständen profitieren, die mehr Lebenspunkte verleihen. Jeder Held ist zu Beginn eines Spieles Stufe 1 kann durch das Sammeln von Erfahrung eine maximale Stufe von 18 erreichen und höchstens 6 Gegenstände gleichzeitig besitzen. Höhere Stufen schalten bessere Fähigkeiten frei und verbessern die Attribute eines Helden (z. B. das Lauftempo).

Abbildung 2.11.: Beispiel eines Charakters der Stufe 1 mit Gegenständen, Fähigkeiten und Gold

Zusammenfassung Zusammenfassend lässt sich sagen, dass jeder Spieler mit seinem Helden durch das Sammeln von Erfahrung und Gegenständen versucht, so stark wie möglich zu werden. Umso stärker die Spieler einer Mannschaft im Vergleich zu der Gegnermannschaft sind, umso leichter ist es diese zu töten und die gegnerische Verteidigung bis hin zum Nexus zu zerstören. In der Tabelle 2.3 sind noch einmal die wichtigsten Begriffe zusammengefasst.

Matchmaking Bevor ein Wettkampfspiel startet, müssen beide Mannschaft ihre Helden auswählen. Jede Mannschaft kann 3 Helden bannen, wie man in Abbildung 2.13 sieht die keiner für dieses Spiel auswählen kann. Nach dieser Bannphase entscheiden sich die Mitglieder jeder Mannschaft abwechselnd für einen Helden. Jeder Held kann dabei nur einmal pro Spiel auftreten. Ziel ist es, eine Zusammenstellung von Helden zu erreichen, die eine gute Synergie, aber auch im Vergleich zur Gegnermannschaft gute Chancen auf einen Sieg haben. Jede Mannschaft besteht dabei aus 5 Rollen: Unterstützung, Mitte, Unten, Oben, Dschungel. Diese Rollen besitzen im Spiel jeweils verschiedene Positionen und Aufgaben.

Begriff:	Bedeutung:
Nexus	Hauptgebäude einer Mannschaft. Wird es zerstört, hat die Mannschaft des zerstörten Nexus verloren.
Held/Charakter	Von einem Spieler gesteuerte Einheit. Jeder Held hat verschiedene Fähigkeiten und Stärken.
Fähigkeiten	Jeder Held hat bestimmte Fähigkeiten, die z. B. das Lauftempo erhöhen. Fähigkeiten profitieren von Gegenständen und Stufenaufstiegen.
Stufe	Jeder Held startet auf Stufe 1 und kann maximal Stufe 18 erreichen. Stufenaufstiege werden durch das Sammeln von Erfahrung ermöglicht und gewähren zusätzliche Attribute wie z. B. Lebenspunkte.
Rollen	Jeder Spieler einer Mannschaft besitzt eine bestimmte Rolle. Jeder Spieler versucht daher einen Charakter zu wählen, der seine Rolle gut erfüllen kann.
Erfahrung	Erfahrung wird zum Stufenaufstieg benötigt. Stufenaufstiege ermöglichen z. B. die Freischaltung zusätzlicher Fähigkeiten und Attribute wie Lebenspunkte.
Gold	Gold ist die Währung innerhalb eines Spieles und wird zum Kauf von Gegenständen benötigt.
Gegenstände	Gegenstände verstärken die Attribute und Fähigkeiten eines Helden.
Tötungen	Helden/Vasallen können sterben, indem ihre Lebenspunkte auf 0 fallen. Führt dabei ein gegnerischer Held den letzten Schlag aus, bekommt er eine Tötung zugeschrieben und erhält Gold/Erfahrung.
Tode	Stirbt ein Held, kann er für eine bestimmte Zeit nicht am Spielgeschehen teilnehmen und kann z. B. einen Turm nicht verteidigen.
Unterstützungen	Unterstützt ein Held einen anderen Helden bei der Tötung eines gegnerischen Helden, bekommt er eine Unterstützung gut geschrieben. Unterstützungen werden mit Gold/Erfahrung belohnt.
Vasallen	Vasallen sind computergesteuerte Einheiten einer Mannschaft, die in regelmäßigen Abständen erscheinen und in Richtung des gegnerischen Nexus laufen, um diesen zu zerstören. Die Tötungen von Vasallen gewährt Gold/Erfahrung.

Tabelle 2.3.: Wichtige Begriffe in *League of Legends*

Abbildung 2.12.: Beispielauschnitt aus einer Hälfte der Karte. Rot bzw. blau umrandete Objekte in der Karte rechts unten gehören zu den jeweiligen Mannschaften

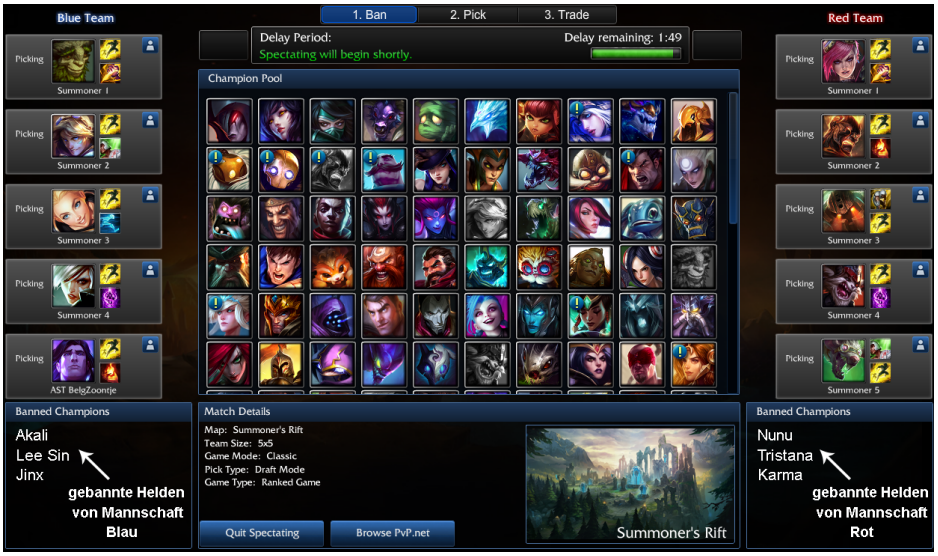

Abbildung 2.13.: Beispiel für die Wahl der Helden mit je 5 ausgewählten und 3 gebannten Helden

Für jeden Sieg bzw. Niederlage bekommt ein Spieler Ranglistenpunkte dazu bzw. abgezogen. Das Ranglistensystem ist in verschiedene Divisionen aufgeteilt, in die Spieler ab einem bestimmten Punktestand auf- bzw. absteigen können. Es kann grundsätzlich zwischen zwei Ranglistenmodi unterschieden werden. Zum einen gibt es die Einzelrangliste: Hier meldet sich ein Spieler für eine Rolle an und wird mit 9 zufällig Spielern (4 in seiner Mannschaft, 5 in der Gegnermannschaft) so zusammengewürfelt, dass alle Rollen besetzt werden. Dabei wird vom System darauf geachtet, dass alle Spieler ungefähr die selbe Spielstärke besitzen, um ein ausgeglichenes Spiel zu ermöglichen. So trifft zum Beispiel ein Anfänger nicht auf einen professionellen Spieler. Der andere Modus ist die Mannschaftsrangliste: Hier treffen

eine Gruppe von 5 Spielern auf eine andere Gruppe von 5 Spielern mit etwa der selben Spielstärke. So wird bei der Einzelrangliste das Können einzelner Spieler bewertet, während bei der Mannschaftsrangliste die Leistung als Mannschaft bewertet wird. In professionellen Turnieren treten immer komplette Mannschaften gegeneinander an.

2.2.2. Riot API

Die Riot *API* wurde Ende 2013 [Ira13] veröffentlicht und steht seitdem jedem mit einem *League of Legends* Konto zur Verfügung. Ihr Sinn ist es, Daten aus *League of Legends* abzurufen. Die *API* antwortet auf Anfragen mit den jeweiligen Daten im JSON Format. Diese JSON Objekte sind unterschiedlich aufgebaut, je nach Art der Abfrage. So sind Abfragen über einen spezifischen Spieler anders aufgebaut als Abfragen über eine Liga, da unterschiedliche Felder benötigt werden. Um Daten von Riot abzurufen wird ein *API-Key* benötigt, den man durch das Einloggen mit einem *League of Legends* Konto erhält. Mit diesem *Development API-Key* lassen sich Anfragen an die *API* stellen, aber maximal 10 Anfragen jede 10 Sekunden bzw. 500 Anfragen jede 10 Minuten. Diese Raten sind für den zu erstellenden Versuchsaufbau völlig ausreichend, sollten aber mehr Anfragen benötigt werden, kann man sich für einen *Production API-Key* bewerben.

Anfrageparameter

Anfragen an die *API* sind beispielsweise folgendermaßen aufgebaut:

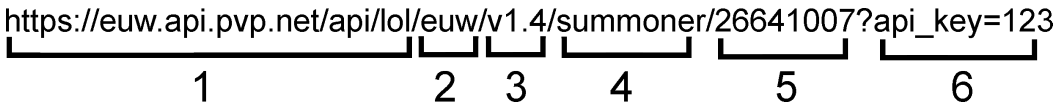

Abbildung 2.14.: Beispielabfrage

Dabei stehen die Abschnitte 1 bis 6 für:

1 Basisteil der URL.

2 Server, auf dem der Spieler ist.

3 Version der *API*.

4 Art der Abfrage.

5 Identifikationsnummer des Spielers.

6 *API-Key*.

Jeder der Abschnitte, bis auf den 5., ist bei jeder der verschiedenen Anfragetypen vorhanden. Je nach Art der Abfrage können noch Felder hinzukommen, wie in diesem Fall die Identifikationsnummer, weil hier Daten über einen bestimmten Spieler abgefragt werden und diese über eindeutige Nummern identifiziert werden. Außerdem wird bei jeder Anfrage ein Statuscode gesendet, der im Falle eines Fehlers einen Indikator gibt, warum die Anfrage gescheitert ist. So wird wird zum Beispiel der Statuscode 429 gesendet, falls das oben genannte Abfragelimit überschritten wurde [Rio16].

Anfragetypen

Die Riot *API* stellt verschiedene Anfragetypen bereit, die verschiedene Informationen liefern. Im Folgenden werden einige der verfügbaren Anfragen vorgestellt, die später auch für den Versuchsaufbau verwendet werden.

Ligen Ligen, hier insbesondere die *Challengerliga* (bestehend aus den 200 besten Spielern einer Region), können mit folgender *URL* abgerufen werden:

- `/api/lol/{region}/v2.5/league/challenger`

Es gibt mehrere Server in *League of Legends*, darum muss in `region` konfiguriert werden, welcher abgerufen werden soll. Im Versuchsaufbau stammen alle Anfragen aus der `euw`-Region.

Spielerstatistik Allgemeine Spielerstatistiken über die Wettkampfspiele einer Saison eines bestimmten Spielers werden mit dieser *URL* abgerufen:

- `/api/lol/{region}/v1.3/stats/by-summoner/{summonerId}/ranked`

Jeder Spieler hat eine eindeutige Identifikationsnummer, die im Feld `summonerId` angegeben werden muss. Als `summonerId` wird im Versuchsaufbau die `26641007` dienen. Dieser Spieler wurde gewählt, weil er oft spielt und somit der Stichprobenumfang groß ist.

Letzte Spiele Informationen über die letzten Spiele eines Spieler werden mit folgenden *URLs* abgerufen:

- `/api/lol/region/v2.2/matchlist/by-summoner/summonerId`
- `/api/lol/region/v1.3/game/by-summoner/summonerId/recent`

Beispielanfrage

Im Folgenden wird einer der oben genannten Anfragen durchgeführt und das Ergebnis in gekürzter Form gezeigt. Mit dieser *URL* wird eine Statistik für die Wettkampfspiele eines Spielers abgerufen:

```
https://euw.api.pvp.net/api/lol/euw/v1.3/stats/by-summoner/26641007/ranked
?season=SEASON2016&api_key=dcb9051e-b562-41bd-b7c4-71eaea6d51ca
```

Listing 2.37: Beispielanfrage an die Riot *API*

Die Abfrage liefert folgendes Ergebnis:

```
{
"summonerId":26641007,"modifyDate":1480315665000,"champions":[
  {"id":110,"stats":{"totalSessionsPlayed":1,"totalSessionsLost":0,
   "totalSessionsWon":1,[...]}},
[...]
  {"id":31,"stats":{"totalSessionsPlayed":3,"totalSessionsLost":1,
   "totalSessionsWon":2,[...]}}]
}
```

Listing 2.38: Beispielantwort der Riot *API*

Wie man an den eckigen Klammern erkennt, werden alle Daten über Charaktere (hier identifiziert über die Identifikationsnummer `summonerId`) in einem Feld gespeichert. So ist zum Beispiel zu sehen, dass der Spieler den Charakter mit der Nummer 31 3 mal gespielt hat (`totalSessionsPlayed`) und davon 2 mal gewonnen hat (`totalSessionsWon`).

2.2.3. Eignung für den ELK-Stack

Der folgenden Abschnitt beschäftigt sich mit der Frage, ob die *League of Legends* Daten als Anwendungsbeispiel für den *ELK-Stack* geeignet sind.

Verfügbarkeit Die Spieldaten lassen sich mit der Riot *API* in genügend kurzen Intervallen abrufen. Dafür wird lediglich ein *League of Legends* Konto benötigt, das kostenlos erstellt werden kann.

Struktur Mit der Riot *API* abgerufene Daten liegen im JSON Format vor. Logstash ist in der Lage diese abzufragen und mit ihnen zu arbeiten. Die verschiedenen Daten wie Zahlen, Strings oder Felder können erkannt und wenn nötig auch bearbeitet werden. So enthalten die Datensätze eindeutige Identifikationsmöglichkeiten wie die Identifikationsnummer eines Spielers, was zum Beispiel zur Duplikatserkennung dienen kann.

Analysierbarkeit Da der Versuchsaufbau testen soll, ob der *ELK-Stack* zur Datenanalyse und Visualisierung geeignet ist, ist es nötig, dass die Daten des Anwendungsbeispiel dies auch ermöglichen. Einige der abrufbaren Daten wie statische Informationen über Charaktere, wie die Namen von Fähigkeiten, sind nicht relevant, da sich daraus wenig aussagekräftige Visualisierungen erstellen lassen. Aber es gibt auch interessantere Datensätze. So sind unter anderem folgende Datensätze interessant, weil mit ihnen Fragen beantwortet werden können wie: Was ist der beste Charakter eines Spielers? Wird ein Spieler im Lauf der Zeit besser? Was machen erfolgreiche Spieler anders als weniger erfolgreiche?

- Ranglisten (mit Namen, Anzahl Siege/Niederlagen und Ligapunkte der Spieler).
- Statistiken über bestimmte Spieler (Liga des Spielers, Anzahl Siege/Niederlagen mit Charakteren, gewählte Rollen)
- Daten über spezifische Spiele (Ausgang des Spieles, Daten über beteiligte Spieler/Charaktere wie Anzahl Tode oder erhaltenes Gold)

So lassen sich Kennzahlen wie Siegwahrscheinlichkeiten, die besten Mannschaftszusammenstellungen, aber auch die Entwicklung der besten 10 Spieler eines Server zum Beispiel anhand eines Liniendiagramms über die Zeit hinweg betrachten. Die Daten ermöglichen das Erstellen vieler der in Kibana bereitgestellten Visualisierungen.

Folgerung

Die Riot *API* erlaubt das Abrufen verschiedenster Daten. Mit diesen Daten ist es möglich, viele der durch den *ELK-Stack* bereitgestellten Funktionen zu testen. So müssen die Daten nach der Abfrage in Logstash strukturiert werden, um sie anschließend indexiert in Elasticsearch zu speichern. Aus diesen gespeicherten Daten können Visualisierungen und Kennzahlen erstellt werden. Somit ermöglichen die Daten aus *League of Legends* das Testen des *ELK-Stacks* als Methode zur Datenanalyse und Visualisierung.

Zusammenfassung

In den obigen Kapitel wurden die Grundlagen der einzelnen Komponenten des *ELK-Stacks* vermittelt. Des Weiteren wurde ein Verständnis über die für den Versuchsaufbau verwendeten Daten geschaffen, und warum diese für den *ELK-Stack* geeignet sind. Dieses Wissen ist für das folgende Kapitel, das sich mit der Implementierung des Versuchsaufbaus beschäftigt, notwendig.

3. Implementierung

Dieses Kapitel setzt sich mit dem Entwurf und der Implementierung des Versuchsaufbaus auseinander. Im Entwurf werden ein Anforderungskatalog, der Aufbau des *ELK-Stacks* und die für die Implementierung zu erledigende Aufgaben behandelt. Daraufhin wird der Versuchsaufbau mit dem in Kapitel 2 vermitteltem Wissen implementiert, indem zuerst Logstash, dann Elasticsearch und schließlich Kibana konfiguriert werden.

3.1. Entwurf des Versuchsaufbaus

Dieser Abschnitt setzt sich mit den Anforderungen und dem Aufbau des Versuchsaufbaus auseinander. Die entstandenen Anforderungen sind in Zusammenarbeit mit der Firma xdi360 entstanden und sollen dazu dienen, dass nach der Umsetzung in Kapitel 4 bewertet werden kann, ob der *ELK-Stack* eine geeignete Methode zur Analyse und Visualisierung von Daten darstellt.

3.1.1. Anforderungen

An einen Versuchsaufbau können zwei verschiedene Arten von Anforderungen gestellt werden. Die funktionalen Anforderungen beschäftigen sich damit, welche Funktionalitäten ein System besitzen soll. Die nichtfunktionalen Anforderungen definieren die geforderten Qualitätseigenschaften an das ganze System. Dazu zählen zum Beispiel die Leistung, Skalierbarkeit oder Zuverlässigkeit.
An den Versuchsaufbau werden folgende Anforderungen gestellt:

Funktionale Anforderungen

FA 1 Logstash soll Spielerstatistiken mithilfe der Riot *API* in regelmäßigen Abständen abrufen.

FA 2 Die abgerufenen Daten sollen, wenn nötig, gefiltert und strukturiert werden.

FA 3 Eventuell fehlende Informationen sollen hinzugefügt werden.

FA 4 Die Daten sollen einem Index zugeordnet werden, der zum Beispiel den Zeitpunkt der Indexierung beinhaltet.

FA 5 Die Daten sollen an ein Elasticsearchsystem weitergeleitet werden.

FA 6 Elasticsearch soll die von Logstash erhaltenden Daten speichern.

FA 7 Elasticsearch soll Duplikate erkennen und löschen.

FA 8 Such- oder Aggregationsanfragen durch Kibana sollen von Elasticsearch bearbeitet werden.

FA 9 Die in Elasticsearch gespeicherten Indexe sollen in Kibana durchsuch- und abrufbar sein.

FA 10 Aus diesen Indexen sollen sich Visualisierung und statistische Werte erstellen lassen, darunter:

- Balkendiagramm mit Heldenstatistiken
- Kreisdiagramm über Verteilung der Helden in einer bestimmten Rolle
- Durchschnittswerte für Tode, Tötungen und Unterstützungen
- *League of Legends Heatmap*
- Liniendiagramm mit Punkteentwicklung innerhalb einer Liga
- Angabe der Siegwahrscheinlichkeit in Prozent

FA 11 Visualisierungen und statistische Werte sollen in einem individuell anpassbaren *Dashboard* abrufbar sein.

FA 12 Das *Dashboard* soll sich automatisch aktualisieren lassen, falls sich Daten verändert oder hinzugefügt wurden.

FA 13 Daten sollen sich nach Zeit ordnen und darstellen lassen, um Veränderungen der Daten über die Zeit zu analysieren.

Nichtfunktionale Anforderungen

NFA 1 Der Versuchsaufbau soll dazu in der Lage sein, vertikal und horizontal zu skalieren.

NFA 2 Der Ausfall eines oder mehrere Rechner im System soll kompensiert werden können.

NFA 3 Aktionen wie Suchen oder Aggregationen sollen maximal nach 2 Sekunden ihre Ergebnisse liefern.

NFA 4 Das *Dashboard* soll aus dem Münchner Wissenschaftsnetz unter dem Port 5601 erreichbar sein.

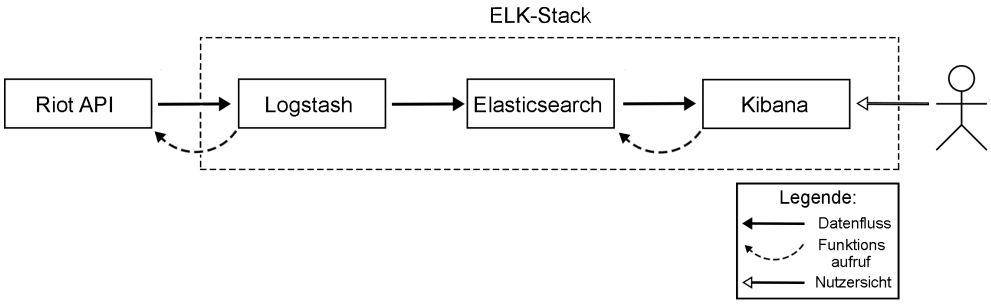

Abbildung 3.1.: Versuchsaufbau

3.1.2. Aufbau

Anhand der aufgestellten Anforderungen und der zu verwendenden Anwendungen lässt sich der Aufbau des Versuchsaufbaus folgendermaßen skizzieren: Der Versuchsaufbau soll aus vier Komponenten bestehen, nämlich der Riot *API*, Logstash, Elasticsearch und Kibana. Dabei lassen sich Logstash, Elasticsearch und Kibana zum ELK-Stack zusammenfassen. Nutzer greifen dann auf Kibana zu, wo das *Dashboard* mit Kennzahlen und Grafiken betrachtet werden kann, aber auch die Daten durchsucht und dem *Dashboard* noch zusätzliche Grafiken hinzufügt werden können. Logstash arbeitet mit Konfigurationsdateien, die an sich aus 3 Komponenten bestehen, nämlich *Input*, *Filter* und *Output*. *Input* ist dafür verantwortlich, Daten aus verschiedensten Quellen, in diesem Fall der Riot *API*, abzurufen. Diese Daten können anschließend im Filterabschnitt gefiltert und strukturiert werden, um daraufhin im Outputabschnitt an ihren Bestimmungsort weitergeleitet zu werden, in diesem Fall Elasticsearch. Nachdem die Daten in Elasticsearch angekommen sind, werden sie indexiert gespeichert und sind nun durchsuchbar bzw. aggregierbar. Kibana nutzt diese Funktionen und stellt den Nutzern die Ergebnisse in Form von Datensätzen oder Grafiken bereit.

3.1.3. Zusammenfassung

Bei der Umsetzung des *ELK-Stacks* müssen folgende Punkte beachtet und je nachdem konfiguriert werden:

Logstash

- Wie können die gewünschten Daten abrufen werden und welche Intervalle sind dafür sinnvoll?
- Welche Informationen beinhalten die abgerufenen Daten?
  ⇒ Fehlende Informationen nachträglich hinzufügen.
- Welche Struktur besitzen die Daten?
  ⇒ Struktur eventuell anpassen.
- Welchem Index sollen die Daten zugeordnet werden?

Elasticsearch

- Wie viele Rechner stehen zur Verfügung?
- Wie viele Primär- und Replikascherben je Index sind sinnvoll?
  ⇒ Abhängig von der Anzahl der Knoten im Netz.

Kibana

- Wie können die Daten gut visualisiert werden?
- Wie lassen sich aus den Daten Rückschlüsse ziehen?
- Wie kann das *Dashboard* übersichtlich gestaltet werden?

Mit diesen Punkten und dessen Umsetzung werden sich die folgenden Kapitel beschäftigen.

3.2. ELK-Stack

In diesem Kapitel werden nacheinander die Bestandteile des *ELK-Stacks* konfiguriert, wie auch in Abbildung 3.1 zu sehen sind. Die Reihenfolge, in der die Bestandteile behandelt werden, entspricht dabei dem Fluss der Daten durch den *Stack* (Logstash ⇒ Elasticsearch ⇒ Kibana).

3.2.1. Konfiguration von Logstash

Dieser Abschnitt handelt von der Erstellung der Konfigurationsdateien für Logstash (Version 2.3.4). Für jede Gruppe von Indexen gibt es eine Konfigurationsdatei. Im Folgenden werden nicht alle Konfigurationsdateien im Detail erklärt, da große Teile redundant wären. Stattdessen werden die wichtigsten Bestandteile vorgestellt.

Input

Da es sich bei der Riot *API* um eine *HTTP API* handelt, ist das *Plugin http-poller* dafür geeignet, Daten von einer bestimmten *URL* abzurufen. Das folgende Beispiel für einen Inputabschnitt ist aus der Konfigurationsdatei `logstashmatchlistsloron.conf`.

```
#Auszug aus logstashmatchlistsloron.conf
input{
  http_poller {
    urls => {
     riotapi => {
      method => get
      url => "https://euw.api.pvp.net/api/lol/euw/v2.5/league/challenger?type=
               RANKED_SOLO_5x5&api_key=dcb9051e-b562-41bd-b7c4-71eaea6d51ca"
                }
            }
    request_timeout => 60
    interval => 86400
    codec => "json"
    metadata_target => "http_poller_metadata"
               }
     }
```

Listing 3.1: Konfiguration des *Inputs*

In diesem Fall wird nur eine *URL* benötigt, es könnten aber theoretisch mehrere angegeben werden. Als Intervall für eine Aktualisierung der Daten wurde 86400 Sekunden, also alle 24 Stunden gewählt, um zum Beispiel bei der Betrachtung der Liga der besten Wettkampfspieler die Entwicklung über die Zeit analysieren zu können, aber auch, um immer möglichst aktuelle Statistiken zu liefern. Das Intervall ist durch die Riot *API* nach unten hin begrenzt, da maximal 10 Anfragen alle 10 Sekunden bzw. 500 Anfragen alle 10 Minuten gestellt werden können. Bei einer Zeitüberschreitung der Anforderung wird im Feld `request timeout` der Standardwert der Funktion (60) beibehalten. `Metadata_target` aktiviert die Speicherung einer Reihe von Metadaten wie den Zeitpunkt des Abrufens unter dem Feld `http_poller_metadata`. Als Format für die Kodierung wird JSON gewählt, da Elasticsearch das JSON Format benötigt. Die Riot *API* antwortet zwar bereits im JSON Format, aber so werden zum Beispiel Felder mit mehreren Objekten in mehrere JSON Dokumente aufgeteilt.

Filter

```
#Auszug aus logstashmatchlistsloron.conf
filter {
  split {
    field => "matches"
        }

  mutate {
    rename => {
      "[matches][matchId]"=> "matchID"
      "[matches][champion]"=> "championID"
              }
          }
 if [championID] == 266 {
    mutate {
    add_field => {
      "championname" => "Aatrox"
              }
          }
                          }
 if [championID] == 103 {
    mutate {
    add_field => {
      "championname" => "Ahri"
              }
          }
                          }
 [...]
 if [championID] == 143 {
    mutate {
    add_field => {
      "championname" => "Zyra"
              }
}}}
```

Listing 3.2: Konfiguration der Filter

Da im bisherigen Dokument alle Spiele eines Spielers in einem einzigen JSON Dokument gespeichert sind, es aber wegen besserer Visualiserungsmöglichkeiten sinnvoller ist, für jedes Spiel ein JSON Dokument anzulegen. Dies durch die `split` Funktion realisiert. Die Funktion

ist in der Lage, String und Array Felder zu splitten, in diesem Fall das Feld `matches`. Jedes der entstanden Resultate ist eine Kopie von dem ursprünglichen Dokument mit dem Unterschied, dass das ausgewählte Feld auf mehrere verteilt wurde. Der `mutate` Filter ermöglicht es, Veränderungen an Feldern durchzuführen. Diese Funktion wird zur Umbenennung von Feldern zur besseren Übersichtlichkeit und Zugriffsmöglichkeit benötigt, da diese vor der Aufteilung *nested fields* waren. *Nested fields* sind Felder von Objekten, die in anderen JSON Objekten verschachtelt gespeichert sind. Der `mutate` Filter kann auch dazu genutzt werden, um Felder hinzuzufügen. Jeder Held in *League of Legends* hat einen bestimmten Namen. Die Riot *API* liefert aber bei Datensätzen nur seine spezifische Identifikationsnummer. Darum ist es zur besseren Leserlichkeit für Menschen sinnvoll, jeder `championID` auch den dazu passenden Namen zuzuordnen. In einer `if`-Anweisungen wird dazu die `championID` abgefragt und falls diese mit einer bestimmten Zahl übereinstimmt, wird das Feld `championname` hinzugefügt mit dem jeweiligen Namen des Helden.

Output

Um die eingelesenen Daten an Elasticsearch und somit auch an Kibana weiterzuleiten, wird die Elasticsearch *Output* Funktion eingesetzt.

```
#Auszug aus logstashmatchlistsloron.conf
output {
 elasticsearch {
   hosts => ["localhost:9200"]
   document_id => "%{matchID}"
   index => "logstashmatchlistsloron"
               }
 stdout { codec => json }
       }
```

Listing 3.3: Konfiguration des Elasticsearch *Output* Funktion in Logstash

Als Hostadresse von Elasticsearch wird `localhost:9200` verwendet, da Logstash auf der selben Maschine wie Elasticsearch ausgeführt wird und auf den `Port 9200` läuft. Um Duplikate zu verhindern, wird jedem Dokument eine Identifikationsnummer hinzugefügt, in diesem Fall die `matchID`. Da diese eindeutig ist, wird, wenn ein neues Dokument mit einer schon vorhandenen Identifikationsnummer dem Index hinzugefügt wird, das alte Dokument durch das neue ersetzt. Duplikate können dadurch entstehen, da die Anfrage die letzten 25 Spiele eines Spielers liefert. Wenn aber ein Spieler bis zur nächste Abfrage noch keine 25 neuen Spiele gemacht hat, würden bereits indexierte Spiele nochmal indexiert werden. Dadurch würden die Daten und somit auch die Analysen verfälscht werden. Jedes Dokument wird außerdem einem Index hinzugefügt. Im folgenden Auszug aus einer Konfigurationsdatei wird dem Index zusätzlich zum Indexnamen noch das Datum der Einspeisung hinzugefügt.

```
#Auszug aus logstashchall.conf
output {
   elasticsearch {
     hosts => ["localhost:9200"]
     index => "logstashchallenger-%{+YYYY.MM.dd}"
                 }
       }
```

Listing 3.4: Konfiguration des Zielindexes

Als weitere Funktion wird `Stdout` verwendet, die dazu nützlich ist, um den *Output* auf Fehler hin zu überprüfen. `Stdout` gibt die ganze Ausgabe auf der Kommandozeile im JSON Format aus. So kann zum Beispiel überprüft werden, ob Felder richtig hinzugefügt werden.

3.2.2. Einrichten von Elasticsearch

Wie in Listing 3.5 zu sehen ist, wurde Elasticsearch (Version 2.3.4) in `Elasticsearch.yml` unter anderem so konfiguriert, dass der Knoten unter `localhost:9200` erreichbar ist.

```
cluster.name: rossp-cluster
node.name: node-rossp-one
network.host: localhost
http.port: 9200
```

Listing 3.5: Ausschnitt aus `Elasticsearch.yml`

Der Rechnerverbund mit dem Namen `rossp-cluster` besteht aus einem Knoten, da dies für den Versuchsaufbaus ausreichend ist. Die Zahl der Primärscherben liegt bei 5 pro Index, was für die maximale Anzahl an durch Logstash erzeugten Dokumenten ausreichend ist. Dies ermöglicht zudem das nachträgliche Hinzufügen von Knoten und somit die horizontale Skalierung. Da bei diesem Versuchsaufbau die Anzahl der Knoten nur eins beträgt, ist es nicht sinnvoll, mehr als eine Replikascherbe pro Primärscherbe zu allokieren. Dies führt dazu, dass im Falle eines Serverausfalls der ganze Rechnerverbund nicht mehr erreichbar wäre. Der Befehl `GET _cluster/health` gibt Informationen über den Status des Rechnerverbundes wieder:

```
{
 "cluster_name": "rossp-cluster",
  "status": "yellow",
  "timed_out": false,
  "number_of_nodes": 1,
  "number_of_data_nodes": 1,
  "active_primary_shards": 251,
  "active_shards": 251,
  "relocating_shards": 0,
  "initializing_shards": 0,
  "unassigned_shards": 251,
  "delayed_unassigned_shards": 0,
  "number_of_pending_tasks": 0,
  "task_max_waiting_in_queue_millis": 0,
  "active_shards_percent_as_number": 50
}
```

Listing 3.6: Beispielausprägung des Rechnerverbundes

Da das *Cluster* nur aus einem Knoten besteht, sind die Replikascherbe inaktiv, was zu einem gelben Status führt. Die Anzahl der Primärscherben wächst jeden Tag, da täglich einer neuer Index mit Daten erstellt wird. Alle bestehende Indexe können mit dem Befehl `GET _cat/indices?v` ausgegeben werden. Im Folgenden ist ein Ausschnitt davon zu sehen:

```
health status index                           pri rep docs store.size
yellow open   logstashchallenger-2016.09.02   5   1   200    188.5kb
yellow open   logstashchallenger-2016.07.28   5   1   200    185.8kb
yellow open   logstashrecentsloron            5   1    10    146.2kb
```

Listing 3.7: Informationen über Indexe

Dabei haben die Spalten folgende Bedeutung:

Tabelle 3.1.: Bedeutung der Spalten in Listing 3.7

`status`	Auf einem geschlossenen Index (`closed`) kann nicht geschrieben werden.
`health`	Status des Rechnerverbundes.
`index`	Name des Index.
`pri`	Anzahl Primärscherben.
`rep`	Anzahl Replikascherben.
`docs`	Anzahl der Dokumente im Index.
`store.size`	Größe eines Index.

Durch eine erhöhte Anzahl von Visualisierungen im *Dashboard* ist die Anzahl der Suchanfragen gestiegen und so wurde bei Abrufen des *Dashboards* der Fehler `Courier Fetch: x of y shards failed` ausgegeben. Dies wurde behoben, indem die `threadpool.search.queue_size` von 1000 auf 5000 erhöht wurde. Die `threadpool.search.queue_size` besagt wie viele Anfragen in der Warteschlange behalten werden und somit nicht verworfen werden[ela16a].

```
PUT /_cluster/settings {
      "transient" : {
          "threadpool.search.queue_size" : 5000
                    }
                       }
```

Listing 3.8: Änderungen der Anzahl der maximalen Anfragen in der Warteschlange

3.2.3. Erstellen von Visualisierungen mit Kibana

Kibana ist so eingerichtet, dass es unter dem Port 5601 mit der IP-Adresse 10.153.99.133 aus dem *MWN* erreichbar ist. Die hier verwendete Version von Kibana ist 4.5.2. Veränderungen zu den Kibana Servereinstellungen können in der Datei `Kibana.yml` durchgeführt werden (siehe Listing 3.9). Dort wurde auch das Elasticsearchcluster definiert, an den die Anfragen von Kibana gesendet werden (`localhost:9200`).

```
server.port: 5601
server.host: "10.153.99.133"
elasticsearch.url: "http://localhost:9200"
```

Listing 3.9: Ausschnitt aus `Kibana.yml`

Bevor mit den Daten gearbeitet werden kann, müssen die in Elasticsearch gespeicherten Indexe zuerst Kibana hinzugefügt werden. Folgende Indexe wurden in Kibana hinzugefügt:

Index:	Daten des Indexes:
`logstashchallenger-*`	Daten über die 200 besten Spiele eines Servers.
`logstashmatchlistsloron`	Informationen über die gespielten Rollen von Sloron.
`logstashrecentsloron`	Informationen über die letzten 10 Spiele von Sloron.
`sloronstats`	Spielerstatistiken der derzeitigen Saison von Sloron.

Tabelle 3.2.: Hinzugefügte Indexe

`Logstashchallenger-*` enthält dabei alle Indexe, deren Namen mit `logstashchallenger-` beginnen (zum Beispiel `logstashchallenger-2017.01.26`). Dadurch müssen täglich erstellte Indexe nicht manuell in Kibana hinzugefügt werden. So können auch Visualisierungen aus ganzen Gruppen von Indexen erstellt werden.

Visualize

Aus den Daten der Indexe können nun Visualisierungen erstellt werden. In den folgenden Abbildungen 3.3 bis 3.6 werden ein Teil der verwendeten Visualisierungen vorgestellt. Dabei wurde versucht, möglichst viele der in Kibana zur Verfügung stehenden Visualisierungen zu verwenden. So bietet es sich für die Darstellung von Informationen über die *Challengerliga* an, Visualisierungen zu verwenden, die das Betrachten der Daten über einen bestimmten Zeitraum hinweg ermöglichen. Bei Spielerstatistiken hingegen eignen sich Visualisierungen, die Verteilungen bei der Wahl von Helden und Rollen darstellen können.

Kreisdiagramm Das Kreisdiagramm mit den Daten aus dem Index `logstashmatchlistsloron` stellt die Verteilung der gespielten Rollen eines bestimmten Spielers dar. Die Größe eines Kreissektors wird durch die Anzahl der Spiele eines Spieler in einer Rolle bestimmt. Jeder dieser Kreissektoren wird danach nochmal unterteilt. Diese Unterteilung sagt aus, welche Helden in einer bestimmten Rolle gespielt wurden.

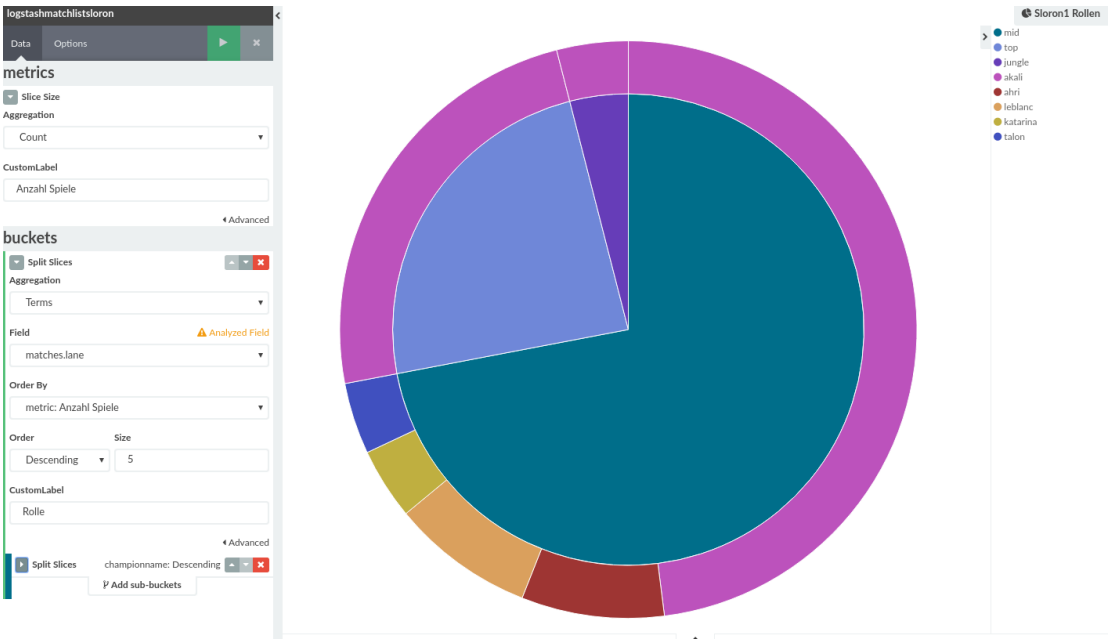

Abbildung 3.2.: Kreisdiagramm: Rollenverteilung von *Sloron*

Radardiagramm Das Radardiagramm ist keine der Standardvisualisierungen, sondern eine der inoffiziellen herunterladbaren Visualisierungen [sir16]. Es besteht aus 3 Achsen. Diese beinhalten die Anzahl der Tode, Tötungen und Unterstützungen. Angezeigt werden die 5 Helden, die am meisten Tode im Durchschnitt besitzen. Dadurch können Rückschlüsse gezogen werden, wie gut ein Spieler einen Charakter beherrscht.

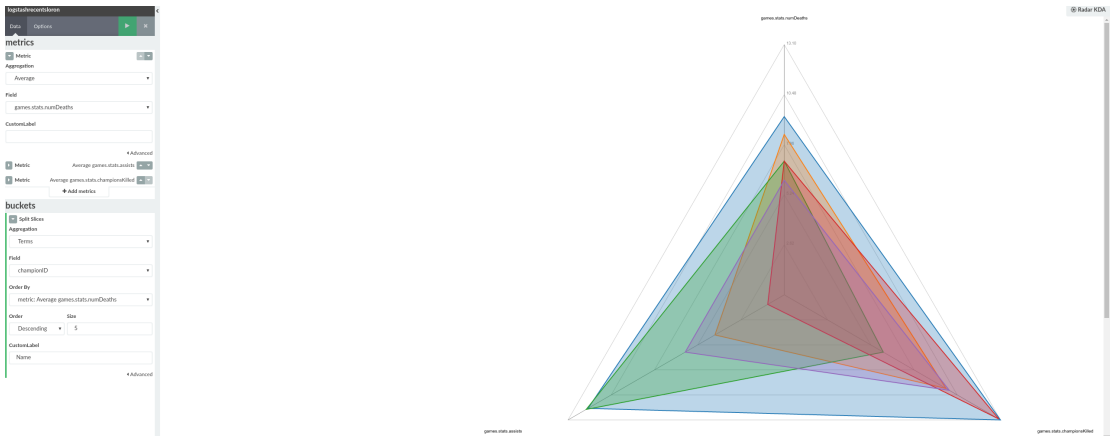

Abbildung 3.3.: Radardiagramm: Verteilung von Toden, Tötungen und Unterstützungen

Flächendiagramm Das Flächendiagramm besteht aus den Indexen `Logstashchallenger-*`. Auf der x-Achse ist die Zeit in Tagen angegeben. Die y-Achse zeigt drei verschiedene Aggregationen. Zum einen das Maximum an Punkten in einer Liga, dann den Durchschnittspunktewert der Liga und schließlich den minimalen Punktewert innerhalb der Liga. Die allgemeine Entwicklung einer Liga wird somit in Form eines Flächendiagramms wiedergegeben, das den Unterschied zwischen den Schlechtesten und Besten einer Liga in Verhältnis mit dem Durchschnitt stellt.

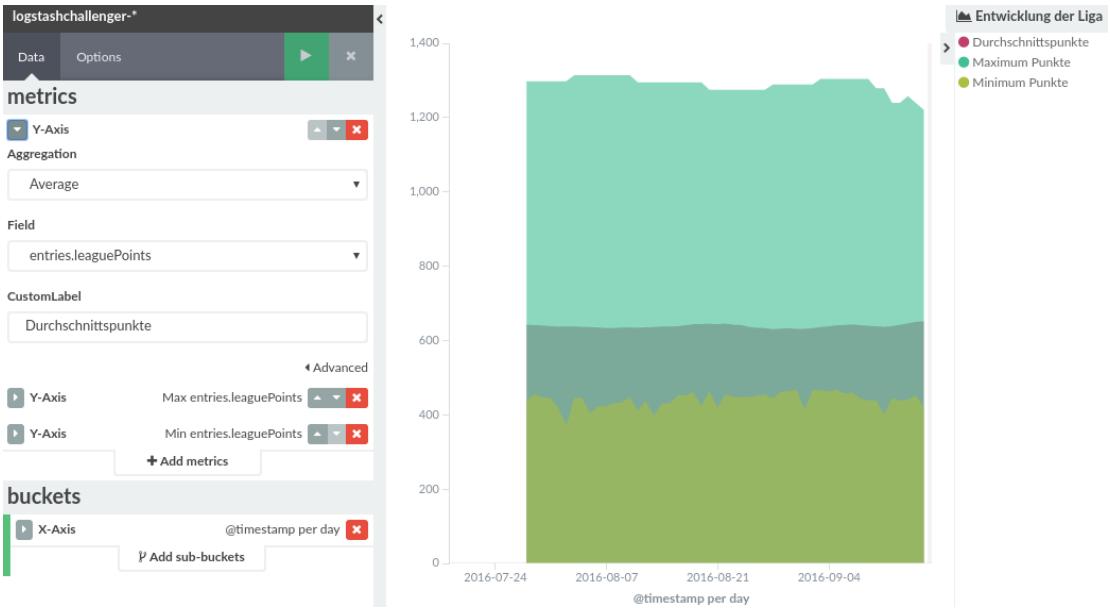

Abbildung 3.4.: Flächendiagramm: Entwicklung der Liga

Liniendiagramm Die Entwicklung der einzelnen Spieler lässt sich in Form eines Liniendiagramm darstellen. Auf der x-Achse gibt es 2 Aggregationen. Auf der untergeordneten wird die Zeit angegeben. Die übergeordnete Aggregation enthält die Identifikationsnummern der

50 besten Spieler, die zu einem Zeitpunkt am meisten Ligapunkte besitzen. Die y-Achse zeigt die Ligapunkte eines Spielers. So lassen sich die 50 besten Spieler über die Zeit hinweg betrachten. Daraus kann geschlussfolgert werden, ob sie z. B. ab einem bestimmten Wert in der Liga stagnieren.

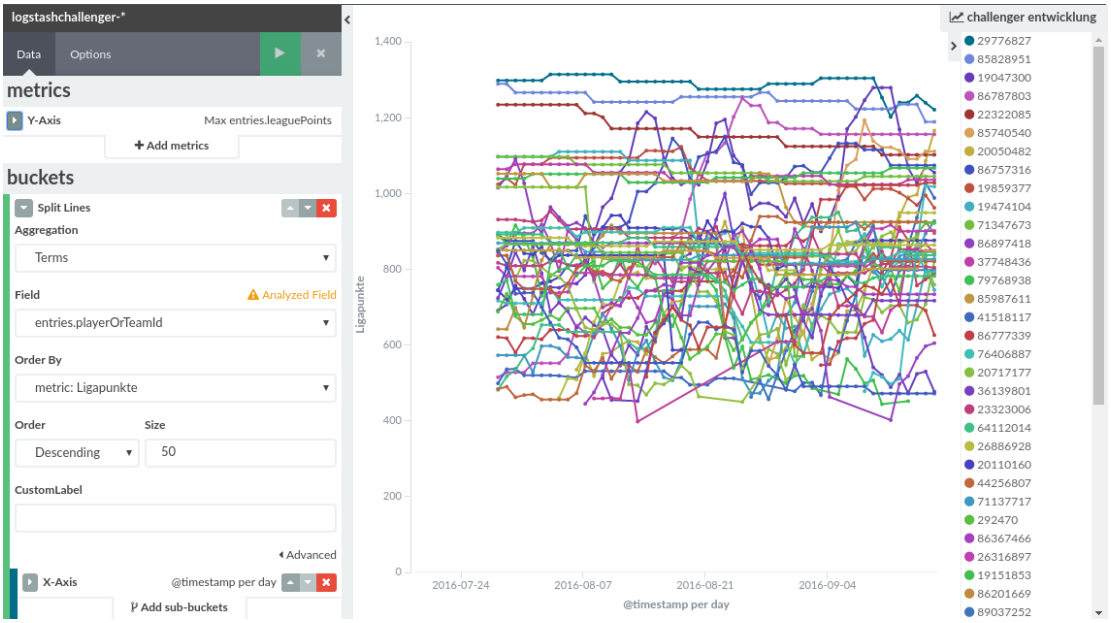

Abbildung 3.5.: Liniendiagramm: Entwicklung der einzelnen Spieler

Metric Die *Metric* aus den Daten von Index `sloronstats` beinhaltet 3 Aggregationen, nämlich die Anzahl aller gespielten Spiele, die Siege und die Niederlagen. Dies gibt Anhaltspunkte über den Umfang der Daten und deren Aussagekraft.

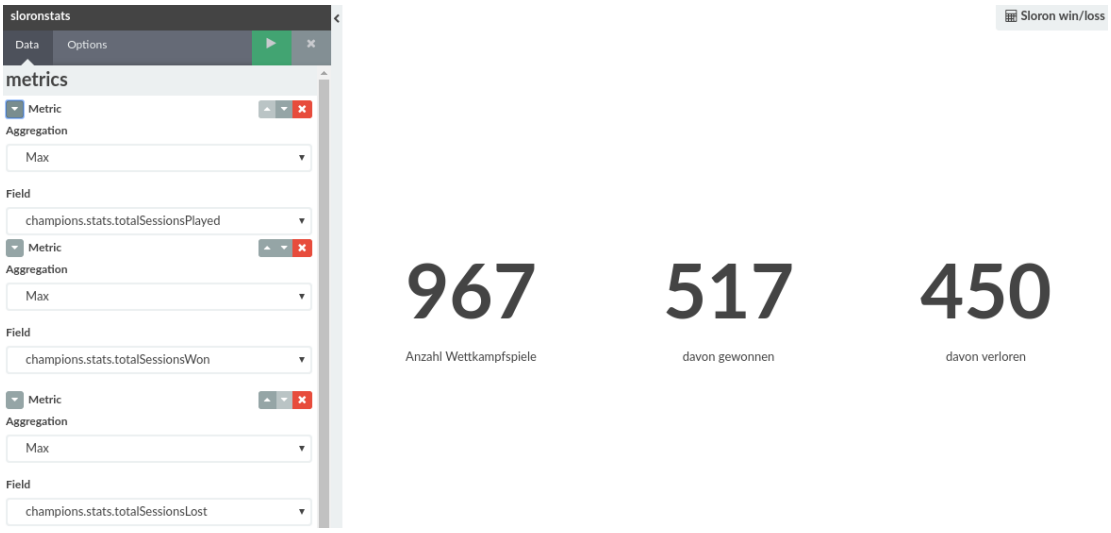

Abbildung 3.6.: Metric: Anzahl Spiele, Siege und Niederlagen von *Sloron*

Säulendiagramm Das Säulendiagramm wird mit den Daten des `sloronstats` Indexes gespeist. Auf der x-Achse finden sich hier die 10 häufigst gespielten Helden in absteigender Reihenfolge wieder. Die y-Achse beinhaltet die Anzahl der Spiele und Siege jedes Helden. Dies lässt auf die Präferenzen eines Spieler hinsichtlich der Heldenwahl schließen, aber auch auf den Erfolg mit den jeweiligen Helden.

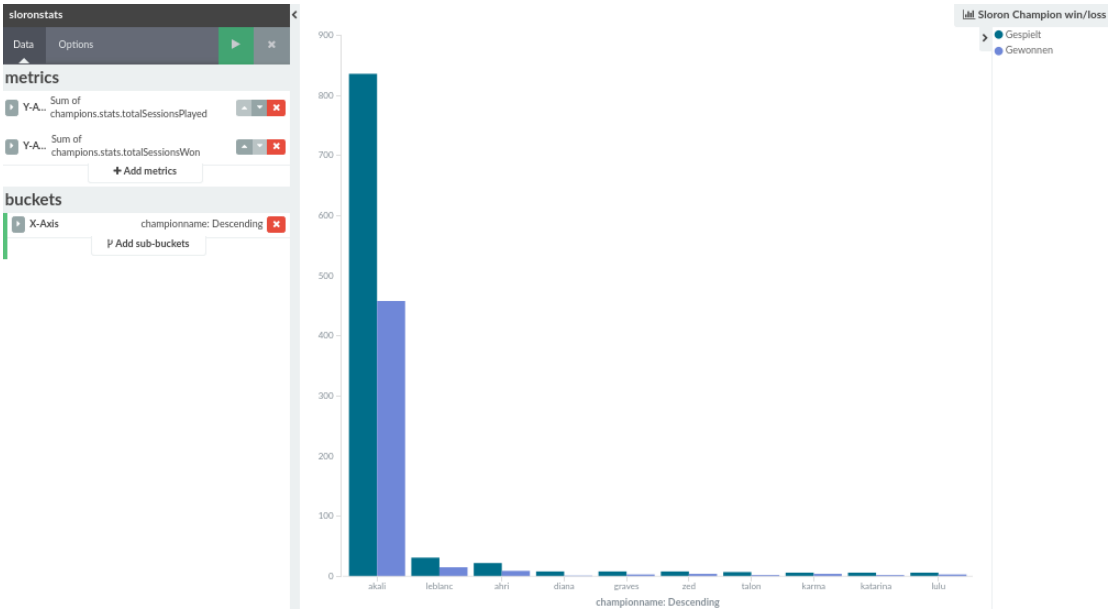

Abbildung 3.7.: Säulendiagramm: Helden Statistik von *Sloron*

Dashboard

Die gespeicherten Visualisierungen können nun einem *Dashboard* (Abbildung 3.8) hinzugefügt werden. Dabei wurden sie so platziert, dass Visualisierungen aus den selbem Kontext in einer logischen Gruppe sind. So gibt eine Gruppe mit Daten über den Spieler Sloron. Die zweite Gruppe von Daten ist die der Liga der besten 200 Spieler eines Servers. An diesen Daten lässt sich beispielsweise beobachten, dass nach einem Patch am 15.9.2016, der die Regeln des Punkteverfalls in der *Challengerliga* verschärft hat [Rio17b], die Durchschnittswerte der Liga gesunken sind. Dies lässt sich damit begründen, da durch die neue Regeln jeder Spieler mindestens ein Spiel am Tag machen muss und sonst 100 Ligapunkte am Tag verliert. Dieses Pensum konnten aber nicht alle Spieler der Liga erfüllen und so haben viele Spieler Ligapunkte verloren, was zur Senkung des Durchschnittswertes führte.
Um mehr der erfassten Daten anzuzeigen, wurde in den Einstellungen der standardmäßige Zeitraum der berücksichtigten von 90 auf 120 Tage erhöht. Die Daten für die Visualisierungen im *Dashboard* können durch Eingabe eines oder mehrerer Suchbegriffe weiter eingegrenzt werden. So werden zum Beispiel durch die Suche `championname:Akali` nur Daten von Spielen verwendet, in denen der Held `Akali` gespielt wurde.

Zusammenfassung

Nach der Implementierung aller Komponenten des *ELK-Stacks* können die Ergebnisse im folgenden Kapitel anhand des Anforderungskataloges bewertet werden.

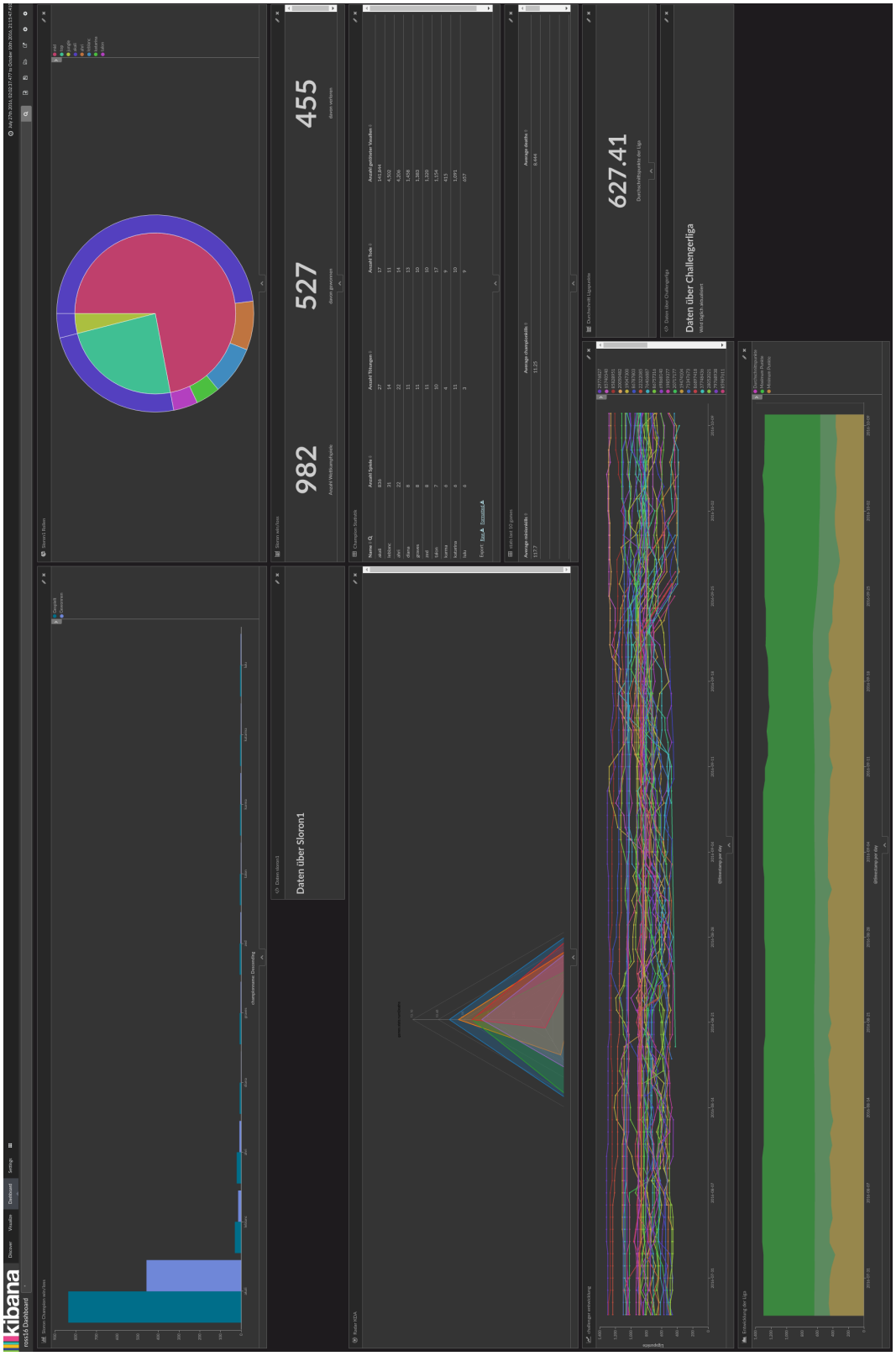

Abbildung 3.8.: Dashboard

4. Fazit

In diesem Kapitel werden die Ergebnisse der Arbeit anhand des Anforderungskataloges bewertet. Den Abschluss dieses Kapitels bildet einen Ausblick auf die Anwendbarkeit des *ELK-Stack* und Ansatzpunkte für Arbeiten an der Thematik.

4.1. Bewertung der Ergebnisse

Im Folgenden wird untersucht, inwieweit der zuvor aufgestellte Anforderungskatalog erfüllt werden konnte.

4.1.1. Funktionale Anforderungen

Die in Kapitel 3 aufgestellten funktionalen Anforderungen konnten folgendermaßen erfüllt werden:

FA 1 Logstash kann die Daten der Riot *API* in regelmäßigen Intervallen (in diesem Fall jede 24 Stunden) abrufen.

FA 2 Die abgerufenen Daten können gefiltert und strukturiert werden, indem z. B. Felder hinzugefügt oder entfernt werden. Lediglich mit verschachtelten Objekten im JSON Format hat Logstash Probleme und das kann nur teilweise mit Filtern behoben werden. Dies führt dazu, dass beim Einlesen von verschachtelten Objekten die Korrelation der Felder von Objekten verloren geht[CG15, Nested Objects], was die spätere Analyse erschwert bzw. das Erkennen von Zusammenhängen unmöglich macht.

FA 3 Fehlende Informationen wie Namen der Charaktere oder Metadaten werden mithilfe von Filtern hinzugefügt.

FA 4 Abgerufene Daten können einem bestehenden oder neuen Index zugeordnet werden, der unter anderem den Zeitpunkt der Indexierung beinhaltet, um Veränderungen über die Zeit hinweg zu analysieren.

FA 5 Logstash leitet die Daten an Elasticsearch über den Port 9200 weiter.

FA 6 Elasticsearch speichert die von Logstash erhaltenen Daten im Rechnerverbund `rossp-cluster`. Bei der Speicherung von Strings gab es aber Probleme, da diese auf drei verschiedene Arten indexiert werden können. Entweder gar nicht, `analyzed` oder `not_analyzed`[CG15, Aggregations and Analysis]. Im Gegensatz zu `not_analyzed` wird bei `analyzed` der String in einzelne *Token* für jedes Wort aufgeteilt. So werden bei Aggregationen nicht der ganze String

aggregiert, sondern lediglich die einzelnen *Tokens*. Dies führt dazu, dass bei der Aggregationen von Charakternamen *Lee Sin* nicht als ein String auftaucht, sondern als zwei Strings, nämlich *Lee* und *Sin*. Das Ändern eines Strings von `analyzed` auf `not_analyzed` ist nach der Indexierung Daten leider nicht mehr möglich. Das bedeutet, dass ein Index komplett neu indexiert werden müsste. Durch eine erneute Indexierung könnten aber z. B. Daten über zeitliche Entwicklung verloren gehen, da die benötigten älteren Daten nicht mehr abgerufen werden können.

FA 7 Mithilfe der in Logstash konfigurierten `document_id` kann Elasticsearch Duplikate erkennen und ersetzt die alten durch die neuen Dokumente.

FA 8 Kibana kommuniziert mit Elasticsearch über Port 9200 und kann damit Such- oder Aggregationsanfragen stellen.

FA 9 Alle in Elasticsearch gespeicherten Indexe sind in Kibana vorhanden und somit durchsuch- und abrufbar sein.

FA 10 Kreis-, Balken- und Liniendiagrammen wurden umgesetzt, wie auch die geforderten Durchschnittswerte. Das Eingeben von mathematischen bzw. statistischen Formel wird derzeit nicht unterstützt, genauso wie eine *Heatmap*. Das Erstellen einer *Heatmap* ist zwar möglich, aber lediglich mit einer Weltkarte und nicht, wie benötigt, mit der Spielkarte aus *League of Legends*. Die Anforderung könnte durch Implementierung eines *Plugins* erfüllt werden. In der verwendeten Version ist zwar im Gegensatz zu früheren Versionen das Verwenden von eigenen *Plugins* schon besser unterstützt, aber immer noch suboptimal. Es kann z. B. dazu kommen, dass es in einem *Plugin* verwendete Funktionen ab einer zukünftigen Version nicht mehr gibt und können neu erschienene Versionen mit viel Anpassungsarbeit verbunden sein, damit ein *Plugin* wieder funktioniert. Das Verwenden von *Plugins* soll ab Kibana 5 besser unterstützt und erleichtert werden.

FA 11 Alle erstellten Visualisierungen und statischen Werte *Dashboard* wurden dem Dashboard `ross16 Dashboard` hinzugefügt.

FA 12 Das *Dashboard* kann so konfiguriert werden, dass es sich in bestimmten Abständen automatisch aktualisiert.

FA 13 Das *Dashboard* zeigt standardmäßig die Daten der letzten 120 Tage an, dies kann aber auch individuell geändert werden.

4.1.2. Nichtfunktionale Anforderungen

An den Versuchsaufbau im Allgemeinen wurden auch nichtfunktionale Anforderungen aufgestellt, die in den folgenden Paragraphen bewertet werden:

NFA 1 Elasticsearch ermöglicht das horizontale und vertikale Skalieren eines Rechnerverbundes. Nach dem Hinzufügen von Knoten würden die Daten automatisch verteilt werden und somit würde eine horizontale Skalierung realisiert werden können. Das System kann auch ohne Probleme vertikal skalieren, indem die Rechenleistung der Rechner erhöht wird.

NFA 2 Obwohl es bei diesem Versuchsaufbau nicht der Fall ist, ist es möglich, ein Rechnerverbund gegen den Ausfall eines oder mehrerer Rechner durch die Konfiguration zusätzlicher Knoten und Replikascherben zu sichern.

NFA 3 Der Cluster beinhaltet über 13000 Dokumente (Dies kann mit dem Befehl `GET _stats` in Sense abgerufen werden) und besitzt bei einer durchschnittlichen Anfragerate von 0,6 pro Sekunde eine durchschnittliche Antwortgeschwindigkeit von 16,12 Millisekunden bzw. eine maximale von 62,17 Millisekunden. Damit erfüllt der Versuchsaufbau das gesetzte Ziel von 2 Sekunden.

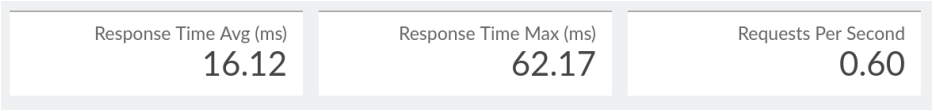

Abbildung 4.1.: Ausschnitt aus dem Statusreiter von Kibana

Im Nachhinein wäre es allerdings effektiver und auch aus Gründen der Übersichtlichkeit besser gewesen, in Elasticsearch mehr Typen zu verwenden und die Anzahl der Indexe zu reduzieren [Gra15].

NFA 4 Kibana ist unter der Adresse `http://10.153.99.133:5601/` aus dem *MWN* erreichbar und das *Dashboard* unter dem Direktlink `http://10.153.99.133:5601/goto/4e0fc532eca9ce71352311b12709bb7f`.

4.1.3. Zusammenfassung

Insgesamt lässt sich sagen, dass der Versuchsaufbau alle Anforderungen vollständig erfüllen konnte bis auf 2, die nur teilweise erreicht werden konnten. Somit stellt der *ELK-Stack* ein geeignetes Tool zum Sammeln, Speichern und Analysieren verschiedener Formen von Daten dar. Es werden regelmäßig neue Versionen und Aktualisierungen veröffentlicht, die Fehler beheben und neue Funktionen bereitstellen. Zur Riot *API* ist zu sagen, dass teilweise Daten in verschachtelten Objekten bereitgestellt werden, die nur eingeschränkt analysierbar sind, da wichtige Korrelationen bei der Spaltung der Objekte verloren gehen. Dies könnte behoben werden, indem jedes Feld einzeln erkannt und einem neuem Objekt zugewiesen wird. Eine weitere Möglichkeit wäre das *Mapping* in Elasticsearch anzupassen [CG15, Nested Objects]. Die Assoziativität des *Dashboards* ist zudem eingeschränkt dadurch, dass die *API* nicht alle benötigten Daten liefert. Beispielsweise wird bei der Abfrage der Charakterstatistiken keine Information über die gewählten Rollen gegeben. So kann keine Erkenntnis darüber gewonnen werden, ob manche Charaktere in bestimmten Rollen effektiver sind als andere.

4.2. Ausblick

Im Folgenden werden die Anwendbarkeit des *ELK-Stacks* und weitere Ansatzpunkte zur Behandlung der Thematik behandelt.

4.2.1. Anwendbarkeit des ELK-Stacks

Zur Anwendbarkeit des *ELK-Stacks* lässt sich sagen, dass Logstash mit der Anfrage und Strukturierung vieler Datentypen zurecht kommt. Elasticsearch ermöglicht die Speicherung großer Datenmengen in einem Verbund mehrerer Rechner und kann somit Ausfallsicherheit und schnelle Antwortzeiten garantieren. Bei den Visualisierungen stehen viele allgemeine Visualisierungen zur Verfügung. Für manche Daten wären allerdings spezielle Visualisierungen erforderlich. Diese selbst zu entwickeln ist zwar möglich, aber umständlich und könnte zum Beispiel durch bessere Unterstützung noch verbessert werden. Falls der *ELK-Stack* sensible Daten verwendet, können diese mit dem *Shield* Paket unter anderem durch Verschlüsselung, Authentifizierung oder Aufzeichnung der System- und Nutzeraktivität geschützt werden.

4.2.2. Weitere Ansatzpunkte

Der entstandene Versuchsaufbau kann in weiteren Arbeiten verbessert oder noch ausgebaut werden durch die Entwicklung zusätzlicher Visualisierungen, die spezifisch auf den Anwendungsfall angepasst sind, in diesem Fall *League of Legends*. Gerade stattfindende Spiele könnten in Echtzeit angezeigt und aufbereitet werden, damit zum Beispiel bei Turnieren die Statistiken im Verlauf des Spiels *live* betrachtet werden können. Dazu müssten alle Konten der Profispieler recherchiert und mit Logstash in kurzen Intervallen abgefragt werden. Mittlerweile kann auf die neue Versionen von Logstash, Elasticsearch und Kibana umgestiegen werden, die unter anderem die Entwicklung von eigenen Visualisierungen besser unterstützt. So könnte die hier nicht realisierte *Heatmap* umgesetzt werden, um wichtige Stellen auf der Spielkarte zu markieren. Auch könnten zu weiteren Evaluation Analysten von professionellen Teams befragt werden, um beurteilen zu können, ob das *Dashboard* für ihren Alltag hilfreich ist und welche Visualisierungen bzw. Daten noch fehlen. Des Weiteren könnte untersucht werden, wie der *ELK-Stack* im Vergleich mit ähnlichen Softwarelösungen wie zum Beispiel Splunk oder Graylog abschneidet.

Abbildungsverzeichnis

2.1. *ELK-Stack* . . . 3
2.2. Rolle von Logstash im *ELK-Stack* [ela16d] . . . 4
2.3. Rolle von Elasticsearch im *ELK-Stack* . . . 10
2.4. Beispiel mit 2 Knoten, 3 Primärscherben mit je einer Replikascherbe[CG15, add_failover] . . . 11
2.5. Beispiel mit 3 Knoten, 3 Primärscherben mit je einer Replikascherben[CG15, scale_horizontally] . . . 11
2.6. Beispiel mit 3 Knoten, 3 Primärscherben mit je zwei Replikascherben[CG15, scale_horizontally] . . . 12
2.7. Einordnung von Kibana im *ELK-Stack* . . . 17
2.8. Übersicht der Funktionen in Kibana . . . 18
2.9. Aktualisieren von Daten in Kibana . . . 21
2.10. Feldbasierters Filtern in Kibana . . . 21
2.11. Beispiel eines Charakters . . . 24
2.12. Beispielauschnitt aus einer Hälfte der Karte . . . 26
2.13. Beispiel für die Wahl der Helden . . . 26
2.14. Beispielabfrage . . . 27

3.1. Versuchsaufbau . . . 33
3.2. Kreisdiagramm: Rollenverteilung von *Sloron* . . . 39
3.3. Radardiagramm: Verteilung von Toden, Tötungen und Unterstützungen . . . 40
3.4. Flächendiagramm: Entwicklung der Liga . . . 40
3.5. Liniendiagramm: Entwicklung der einzelnen Spieler . . . 41
3.6. Metric: Anzahl Spiele, Siege und Niederlagen von *Sloron* . . . 41
3.7. Säulendiagramm: Helden Statistik von *Sloron* . . . 42
3.8. Dashboard . . . 43

4.1. Ausschnitt aus dem Statusreiter von Kibana . . . 47

Literaturverzeichnis

[Ama16] Amazon: *Amazon Elasticsearch Service*. https://aws.amazon.com/de/elasticsearch-service/, 2016. [Online; accessed March 9, 2017].

[CG15] C. Gormley, Z. Tong: *Elasticsearch: The Definitive Guide*. elastic, 2015.

[che16] chernjie: *grok-patterns*. https://github.com/logstash-plugins/logstash-patterns-core/blob/master/patterns/grok-patterns, 2016. [Online; accessed March 9, 2017].

[Cis16] Cisco: *The Zettabyte Era: Trends and Analysis*. Technischer Bericht, Cisco, Juni 2016. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.pdf.

[ela16a] elastic: *Elasticsearch Reference*. https://www.elastic.co/guide/en/elasticsearch/reference/2.3/modules-threadpool.html, 2016. [Online; accessed March 9, 2017].

[ela16b] elastic: *Kibana User Guide*. https://www.elastic.co/guide/en/kibana/4.5/index.html, 2016. [Online; accessed March 9, 20176].

[ela16c] elastic: *Logstash Reference*. https://www.elastic.co/guide/en/logstash/2.3/index.html, 2016. [Online; accessed March 9, 2017].

[ela16d] elastic: *Setting Up an Advanced Logstash Pipeline*. https://www.elastic.co/guide/en/logstash/2.3/advanced-pipeline.html, 2016. [Online; accessed March 9, 2017].

[Eth16] Ethier, Nick: *Grok Debugger*. http://grokdebug.herokuapp.com/, 2016. [Online; accessed March 9, 2017].

[Gra15] Grand, Adrien: *Index vs. Type*. https://www.elastic.co/blog/index-vs-type, 2015. [Online; accessed March 9, 2017].

[Hor16] Horohoe, Chad: *Wikimedia moving to Elasticsearch*. http://blog.wikimedia.org/2014/01/06/wikimedia-moving-to-elasticsearch/, 2016. [Online; accessed March 9, 2017].

[Ira13] Irani, By Romin: *Riot Games API gives access to League of Legends Data*. http://www.programmableweb.com/news/riot-games-api-gives-access-to-league-legends-data/2013/12/18, 2013. [Online; accessed March 9, 2017].

[Mag15] Magus: *Worlds 2015 Viewership*. lolesports, Dezember 2015. http://www.lolesports.com/en_US/articles/worlds-2015-viewership.

[Rio16] RIOT: *API Response Error Codes*. https://developer.riotgames.com/docs/response-codes, 2016. [Online; accessed March 9, 2017].

[Rio17a] RIOT: *Champions*. http://gameinfo.euw.leagueoflegends.com/en/game-info/champions/, 2017. [Online; accessed March 21, 2017].

[Rio17b] RIOT: *Patch 6.18 notes*. http://euw.leagueoflegends.com/en/news/game-updates/patch/patch-618-notes#patch-ranked-decay, 2017. [Online; accessed March 21, 2017].

[sir16] SIREN: *Kibana Radar Chart Plugin*. http://github.com/sirensolutions/kibi_radar_vis, 2016. [Online; accessed March 9, 2017].

[Sto16] STOERR, HANS-PETER: *Grok Constructor*. http://grokconstructor.appspot.com/, 2016. [Online; accessed March 9, 2017].

[Tas14] TASSI, P.: *Riot's 'League of Legends' Reveals Astonishing 27 Million Daily Players, 67 Million Monthly*. Forbes, Januar 2014. http://www.forbes.com/sites/insertcoin/2014/01/27/riots-league-of-legends-reveals-astonishing-27-million-daily-players-67-million-monthly/#43f29c803511.