ELK stack:

Lucene:

文档: Document

包含了一个或多个域的容器;

field:value

域:

有很多选项

索引选项、存储选项、域向量使用选项;

索引选项用于通过倒排索引来控制文本是否可被搜索:

Index:ANYLYZED: 分析(切词)并单独作为索引项;

Index.Not_ANYLYZED: 不分析(不切词), 把整个内容当一个索引项;

Index.ANYLYZED_NORMS: 类似于Index:ANALYZED, 但不存储token的Norms(加权基准)信息;

Index.Not_ANYLYZED_NORMS: 类似于Index:Not_ANALYZED, 但不存储值的Norms(加权基准)信息;

Index.NO: 不对此域的值进行索引; 因此不能被搜索;

存储选项: 是否需要存储域的真实值;

title:This is a Notebook.

store.YES: 存储真实值

store.NO: 不存储真实值

域向量选项用于在搜索期间该文档所有的唯一项都能完全从文档中检索时使用;

文档和域的加权操作

加权计算标准;

搜索:

查询Lucene索引时, 它返回的是一个有序的scoreDoc对象; 查询时, Lucene会为每个文档计算出其score;

API:

IndexSearcher: 搜索索引入口;

Query及其子类:

QueryParser

TopDocs

ScoreDoc

Lucene的多样化查询:

IndexSearcher中的search方法;

TermQuery: 对索引中的特定项进行搜索; Term是索引中的最小索引片段, 每个Term包含了一个域名和一个文本值;

title: This is a Desk.

owner: Tom Blair

description: this is a desk, it's belong to Tom.

title: This is a table.
owner: Clinton
description: this is a desk, it's belong to Clinton.
This: (1) (2)
Desk: (1)
table: (2)

TermRangeQuery：在索引中的多个特定项中进行搜索，能搜索指定的多个域；
NumericRangeQuery：做数值范围搜索；
PrefixQuery：用于搜索以指定字符串开头的项；
BooleanQuery：用于实现组合查询；组合逻辑有：AND, OR, NOT；
PhraseQuery：
WildcardQuery：
FuzzyQuery：模糊查询；Levenshtein

ElasticSearch：

ES是一个基于Lucene实现的开源、分布式、Restful的全文本搜索引擎；此外，它还是一个分布式实时文档存储，其中每个文档的每个field均是被索引的数据，且可被搜索；也是一个带实时分析功能的分布式搜索引擎，能够扩展至数以百计的节点实时处理PB级的数据。

基本组件：

索引(index)：文档容器，换句话说，索引是具有类似属性的文档的集合。类似于表。索引名必须使用小写字母；

类型(type)：类型是索引内部的逻辑分区，其意义完全取决于用户需求。一个索引内部可定义一个或多个类型。一般来说，类型就是拥有相同的域的文档的预定义。

文档(document)：文档是Lucene索引和搜索的原子单位，它包含了一个或多个域。是域的容器；基于JSON格式表示。
每个域的组成部分：一个名字，一个或多个值；拥有多个值的域，通常称为多值域；

映射(mapping)：原始内容存储为文档之前需要事先进行分析，例如切词、过滤掉某些词等；映射用于定义此分析机制该如何实现；除此之外，ES还为映射提供了诸如将域中的内容排序等功能。

ES的集群组件：

Cluster：ES的集群标识为集群名称；默认为"elasticsearch"。节点就是靠此名字来决定加入到哪个集群中。一个节点只能属性于一个集群。

Node：运行了单个ES实例的主机即为节点。用于存储数据、参与集群索引及搜索操作。节点的标识靠节点名。

Shard：将索引切割成为的物理存储组件；但每一个shard都是一个独立且完整的索引；创建索引时，ES默认将其分割为5个shard，用户也可以按需自定义，创建完成之后不可修改。

shard有两种类型：primary shard和replica。Replica用于数据冗余及查询时的负载均衡。每个主shard的副本数量可自定义，且可动态修改。

ES Cluster工作过程：

启动时，通过多播(默认)或单播方式在9300/tcp查找同一集群中的其它节点，并与之建立通信。

集群中的所有节点会选举出一个主节点负责管理整个集群状态，以及在集群范围内决定各shards的分布方式。站在用户角度而言，每个均可接收并响应用户的各类请求。

集群有状态：green, red, yellow

官方站点：https://www.elastic.co/

JDK：
Oracle JDK
OpenJDK

ES的默认端口：
参与集群的事务：9300/tcp
transport.tcp.port
接收请求：9200/tcp
http.port

Restful API：
四类API：
(1) 检查集群、节点、索引等健康与否，以及获取其相应状态；
(2) 管理集群、节点、索引及元数据；
(3) 执行CRUD操作；
(4) 执行高级操作，例如paging, filtering等；

ES访问接口：9200/tcp

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

VERB: GET, PUT, DELETE等；
PROTOCOL: http, https
QUERY_STRING：查询参数，例如?pretty表示用易读的JSON格式输出；
BODY: 请求的主体；

回顾：搜索引擎和ES

搜索引擎：
索引组件、搜索组件

索引：倒排索引

索引组件：Lucene
搜索组件：ES

数据获取组件：solr, Nutch, Grub, Apeture

ES：
索引(index)，类型(type)，文件(document)，映射(mapping)

集群(cluster)，节点(node)，shard(primary, replica)
9300/tcp

每个索引的分片数量：5
每个分片也应该会副本：1

jvm：oracle jdk, openjdk

用户访问接口：9200/tcp

Restful：
API

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

_cat API：

ELK stack(2)

Cluster APIs:

health:
curl -XGET 'http://172.16.100.67:9200/_cluster/health?pretty'

state：
curl -XGET 'http://172.16.100.67:9200/_cluster/state/<metrics>?pretty'

stats：
curl -XGET 'http://172.16.100.67:9200/_cluster/stats'

节点状态：
curl -XGET 'http://172.16.100.67:9200/_nodes/stats'

Plugins：

插件扩展ES的功能：
添加自定义的映射类型、自定义分析器、本地脚本、自定义发现方式；

安装：
直接将插件放置于plugins目录中即可；
使用plugin脚本进行安装；

/usr/share/elasticsearch/bin/plugin -h
-l
-i, --install
-r, --remove

站点插件：
http://HOST:9200/_plugin/plugin_name

CRUD操作相关的API：

创建文档：
curl -XPUT 'localhost:9200/students/class1/2?pretty' -d '
> {
> "first_name": "Rong",
> "last_name": "Huang",
> "gender": "Female",
> "age": 23,
> "courses": "Luoying Shenjian"
> }'

```
{
  "_index" : "students",
  "_type" : "class1",
  "_id" : "2",
  "_version" : 1,
  "created" : true
}
```

获取文档:

```
~]# curl -XGET 'localhost:9200/students/class1/2?pretty'
{
  "_index" : "students",
  "_type" : "class1",
  "_id" : "2",
  "_version" : 1,
  "found" : true,
  "_source":
{
  "first_name": "Rong",
  "last_name": "Huang",
  "gender": "Female",
  "age": 23,
  "courses": "Luoying Shenjian"
}
}
```

更新文档:

PUT方法会覆盖原有文档;

如果只更新部分内容，得使用_update API

```
~]# curl -XPOST 'localhost:9200/students/class1/2/_update?pretty' -d '
{
  "doc": { "age": 22 }
}'
{
  "_index" : "students",
  "_type" : "class1",
  "_id" : "2",
  "_version" : 2
}
```

删除文档:

DETELE

```
~]# curl -XDELETE 'localhost:9200/students/class1/2'
```

删除索引:

```
~]# curl -XDELETE 'localhost:9200/students'

~]# curl -XGET 'localhost:9200/_cat/indices?v'
```

查询数据:

Query API

Query DSL：JSON based language for building complex queries.

用户实现诸多类型的查询操作，比如，simple term query, phrase, range boolean, fuzzy等；

ES的查询操作执行分为两个阶段：
分散阶段：
合并阶段：

查询方式：
向ES发起查询请求的方式有两种：
1、通过Restful request API查询，也称为query string；
2、通过发送REST request body进行；

```
~]# curl -XGET 'localhost:9200/students/_search?pretty'

~]# curl -XGET 'localhost:9200/students/_search?pretty' -d '
> {
>   "query": { "match_all": {} }
> }'
```

多索引、多类型查询：

/_search：所有索引；
/INDEX_NAME/_search：单索引；
/INDEX1,INDEX2/_search：多索引；
/s*,t*/_search：
/students/class1/_search：单类型搜索
/students/class1,class2/_search：多类型搜索

Mapping和Analysis：
ES：对每一个文档，会取得其所有域的所有值，生成一个名为“_all”的域；执行查询时，如果在query_string未指定查询的域，则在_all域上执行查询操作；

```
GET /_search?q='Xianglong'
GET /_search?q='Xianglong%20Shiba%20Zhang'
GET /_search?q=courses:'Xianglong%20Shiba%20Zhang'
GET /_search?q=courses:'Xianglong'
```

前两个：表示在_all域搜索；
后两个：在指定的域上搜索；

数据类型：string, numbers, boolean, dates

查看指定类型的mapping示例：
```
~]# curl 'localhost:9200/students/_mapping/class1?pretty'
```

ES中搜索的数据广义上可被理解为两类：
types:exact
full-text

精确值：指未经加工的原始值；在搜索时进行精确匹配；
full-text：用于引用文本中数据；判断文档在多大程序上匹配查询请求；即评估文档与用户请求查询的相关度；

为了完成full-text搜索，ES必须首先分析文本，并创建出倒排索引；倒排索引中的数据还需进

行“正规化”为标准格式；
分词
正规化

即分析

分析需要由分析器进行：analyzer

分析器由三个组件构成：字符过滤器、分词器、分词过滤器

ES内置的分析器：
Standard analyzer：
Simple analyzer
Whitespace analyzer
Language analyzer

分析器不仅在创建索引时用到；在构建查询时也会用到；

Query DSL：
request body：

分成两类：
query dsl：执行full-text查询时，基于相关度来评判其匹配结果；
查询执行过程复杂，且不会被缓存；
filter dsl：执行exact查询时，基于其结果为“yes”或“no”进行评判；
速度快，且结果缓存；

查询语句的结构：

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

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

filter dsl：

term filter：精确匹配包含指定term的文档；

```
{ "term": {"name": "Guo"} }

curl -XGET 'localhost:9200/students/_search?pretty' -d {
    "query": {
        "term": {
            "name": "Guo"
        }
    }
}
```

terms filter：用于多值精确匹配；
{ "terms": { "name": ["Guo", "Rong"] }}

range filters：用于在指定的范围内查找数值或时间。
{ "range":
"age": {
"gte": 15,
"lte": 25
}
}

gt, lt, gte, lte

exists and missing filters：
{
"exists": {
"age": 25
}
}

boolean filter：
基于boolean的逻辑来合并多个filter子句；
must：其内部所有的子句条件必须同时匹配，即and；
must: {
"term": { "age": 25 }
"term": { "gender": "Female" }
}

must_not：其所有子句必须不匹配，即not
must_not: {
"term": { "age": 25 }
}

should：至少有一个子句匹配，即or
should: {
"term": { "age": 25 }
"term": { "gender": "Female" }
}

QUERY DSL：

match_all Query：
用于匹配所有文档，没有指定任何query，默认即为match_all query.
{ "match_all": {} }

match Query：
在几乎任何域上执行full-text或exact-value查询；

如果执行full-text查询：首先对查询时的语句做分析；
{ "match": {"students": "Guo" }}

如果执行exact-value查询：搜索精确值；此时，建议使用过滤，而非查询；
{ "match": {"name": "Guo"} }

multi_match Query：
用于在多个域上执行相同的查询；

```
{
    "multi_match":
        "query": full-text search
        "field": {'field1', 'field2'}
}

{
    "multi_match":
        "query": {
            "students": "Guo"
        }
        "field":
            {
                "name",
                "description"
            }
}
```

bool query:

基于boolean逻辑合并多个查询语句；与bool filter不同的是，查询子句不是返回"yes"或"no"，而是其计算出的匹配度分值。因此，boolean Query会为各子句合并其score；

must:
must_not:
should:

合并filter和query:

```
{
    "filterd": {
        query: { "match": {"gender": "Female"} }
        filter: { "term": {"age": 25}}
    }
}
```

查询语句语法检查:

```
GET /INDEX/_validate/query?pretty
{
    ...
}


GET /INDEX/_validate/query?explain&pretty
{
    ...
}
```

ELK stack的另外两个组件:

L: logstash
K: Kibina

Logstash:

支持多数据获取机制，通过TCP/UDP协议、文件、syslog、windows EventLogs及STDIN等；获取到数据后，它支持对数据执行过滤、修改等操作；

JRuby语言，JVM；

agent/server

配置框架：

```
input {
    ...
}

filter {
    ...
}

output {
    ...
}
```

四种类型的插件：

input, filter, codec, output

数据类型：

Array：[item1, item2,...]
Boolean：true, false
Bytes：
Codec：编码器
Hash：key => value
Number：
Password：
Path：文件系统路径;
String：字符串

字段引用：[]

条件判断：

==, !=, <, <=, >, >=
=~, !~
in, not in
and, or
()

回顾：ES查询、Logstash

ES查询：query-string，Query DSL

```
q=class:"Huashan"
```

CRUD：

curl -XPUT, -XGET, -XPOST, -XDELETE

_validate

Logstash：数据收集，日志数据;

高度插件化：

input, codec, filter, output

ELK stack(3)

Logstash的工作流程：input | filter | output, 如无需对数据进行额外处理，filter可省略;

```
input {
        stdin {}
}

output {
        stdout {
                codec   => rubydebug
        }
}
```

Logstash的插件：

input插件：

File：从指定的文件中读取事件流；
使用FileWatch（Ruby Gem库）监听文件的变化。
.sincedb：记录了每个被监听的文件的inode, major number, minor nubmer, pos;

```
input {
   file {
      path => ["/var/log/messages"]
      type => "system"
      start_position => "beginning"
   }
}

output {
   stdout {
      codec   => rubydebug
   }
}
```

udp：通过udp协议从网络连接来读取Message，其必备参数为port，用于指明自己监听的端口，host则用指明自己监听的地址；

collectd：性能监控程序；

CentOS 7:

epel源：

yum install collectd

配置文件/etc/collectd.conf

```
Hostname    "node3.magedu.com"
LoadPlugin syslog
LoadPlugin cpu
LoadPlugin df
LoadPlugin interface
LoadPlugin load
LoadPlugin memory
LoadPlugin network
<Plugin network>
        <Server "172.16.100.70" "25826">
        # 172.16.100.70是logstash主机的地址，25826是其监听的udp端口；
```

```
        </Server>
    </Plugin>
    Include "/etc/collectd.d"

systemctl start collectd.service
```

logstash端:

```
input {
    udp {
        port    => 25826
        codec   => collectd {}
        type    => "collectd"
    }
}

output {
    stdout {
        codec   => rubydebug
    }
}
```

redis插件:

从redis读取数据，支持redis channel和lists两种方式;

filter插件:

用于在将event通过output发出之前对其实现某些处理功能。grok

grok：用于分析并结构化文本数据；目前 是logstash中将非结构化日志数据转化为结构化的可查询数据的不二之选。

syslog, apache, nginx

模式定义位置：/opt/logstash/vendor/bundle/jruby/1.9/gems/logstash-patterns-core-0.3.0/patterns/grok-patterns

语法格式:

%{SYNTAX:SEMANTIC}

SYNTAX：预定义模式名称;

SEMANTIC：匹配到的文本的自定义标识符;

1.1.1.1 GET /index.html 30 0.23

%{IP:clientip} %{WORD:method} %{URIPATHPARAM:request} %{NUMBER:bytes} %{NUMBER:duration}

```
input {
    stdin {}
}

filter {
    grok {
        match => { "message" => "%{IP:clientip} %{WORD:method} %{URIPATHPARAM:request} %{NUMBER:bytes} %{NUMBER:duration}" }
    }
}
```

```
output {
   stdout {
      codec   => rubydebug
   }
}
```

```
{
      "message" => "1.1.1.1 GET /index.html 30 0.23",
     "@version" => "1",
   "@timestamp" => "2015-11-25T02:13:52.558Z",
         "host" => "node4.magedu.com",
     "clientip" => "1.1.1.1",
       "method" => "GET",
      "request" => "/index.html",
        "bytes" => "30",
     "duration" => "0.23"
}
```

自定义grok的模式:

grok的模式是基于正则表达式编写，其元字符与其它用到正则表达式的工具awk/sed/grep/pcre差别不大。

```
PATTERN_NAME (?the pattern here)
```

匹配apache log

```
input {
   file {
      path    => ["/var/log/httpd/access_log"]
      type    => "apachelog"
      start_position => "beginning"
   }
}

filter {
   grok {
      match => { "message" => "%{COMBINEDAPACHELOG}" }
   }
}

output {
   stdout {
      codec   => rubydebug
   }
}
```

nginx log的匹配方式:

将如下信息添加至 /opt/logstash/vendor/bundle/jruby/1.9/gems/logstash-patterns-core-0.3.0/patterns/grok-patterns文件的尾部:

```
NGUSERNAME [a-zA-Z\.\@\-\+_%]+
NGUSER %{NGUSERNAME}
NGINXACCESS %{IPORHOST:clientip} - %{NOTSPACE:remote_user} \[%{HTTPDATE:timestamp}\] \"(?:%{WORD:verb} %{NOTSPACE:request}(?: HTTP/%{NUMBER:httpversion})?|%{DATA:rawrequest})\" %{NUMBER:response} (?:%{NUMBER:bytes}|-)
```

```
%{QS:referrer} %{QS:agent} %{NOTSPACE:http_x_forwarded_for}
                        input {
                            file {
                                    path    => ["/var/log/nginx/access.log"]
                                    type    => "nginxlog"
                                    start_position => "beginning"
                            }
                        }

                        filter {
                            grok {
                                    match => { "message" => "%{NGINXACCESS}" }
                            }
                        }

                        output {
                            stdout {
                                    codec   => rubydebug
                            }
                        }
```

output插件:

```
        stdout {}

        elasticsearch {

        }
```