

# 高吞吐ELK实践

---

吴晓刚@携程旅行网 2015.10.25

# 携程ES集群规模- 30个数据结点

## Cluster Overview

**8,012**  
Total Shards

**8,012**  
Successful Shards

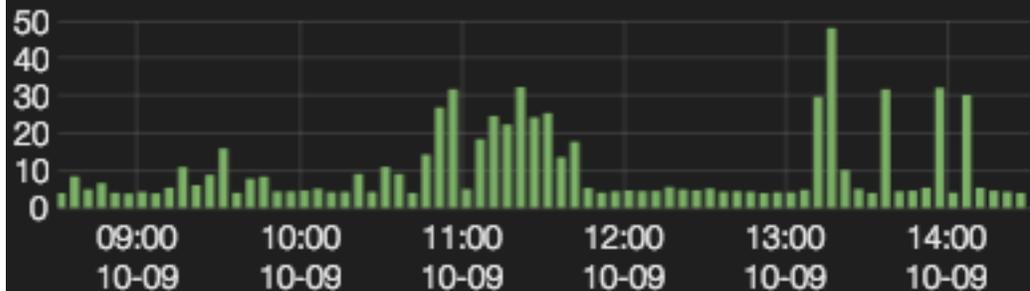
**597**  
Indices

**159,231,461,025**  
Documents

**50.7TB**  
Size

### ES API RESPONSE TIME

View ▶ | 🔍 Zoom Out | ● url:\*\_search\* (30320) responsetime max per 5m | (30320 hits)



### ES API RESPONSE TIME PERCENTILE

**2.180** s (90.0)

Query	50.0%	75.0%	90.0%	95.0%	99.0%
● url:*_search*	0.059 s	0.599 s	2.180 s	3.980 s	11.038 s

# 议程

---

- 监控、监控、监控
- 系统架构
- 写入吞吐量如何提升
- 优化查询、聚合速度

# 监控API

## Nodes Stats:

- GET `/_nodes/stats`
- GET `/_nodes/nodeID1,nodeID2/stats`

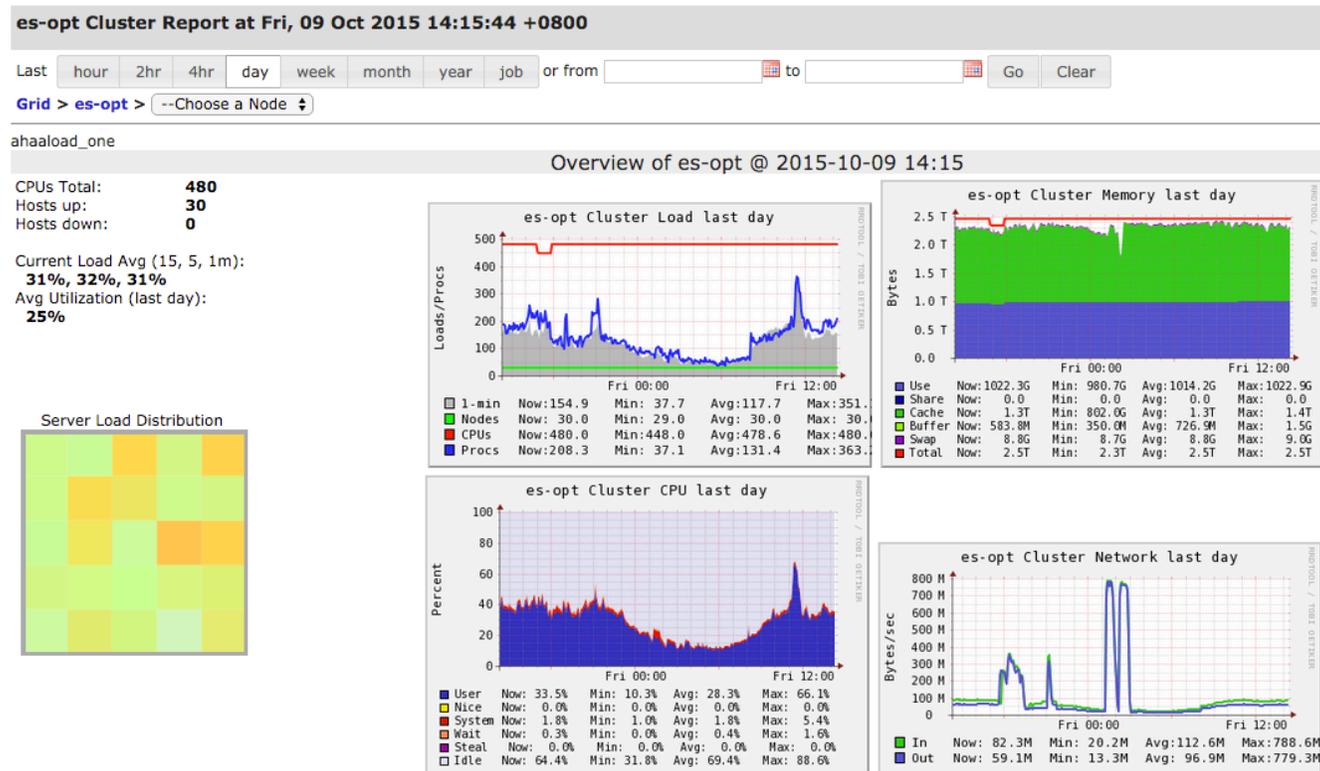
## CAT:

- GET `/_cat/health`
- GET `/_cat/nodes`
- .....

## Plugins:

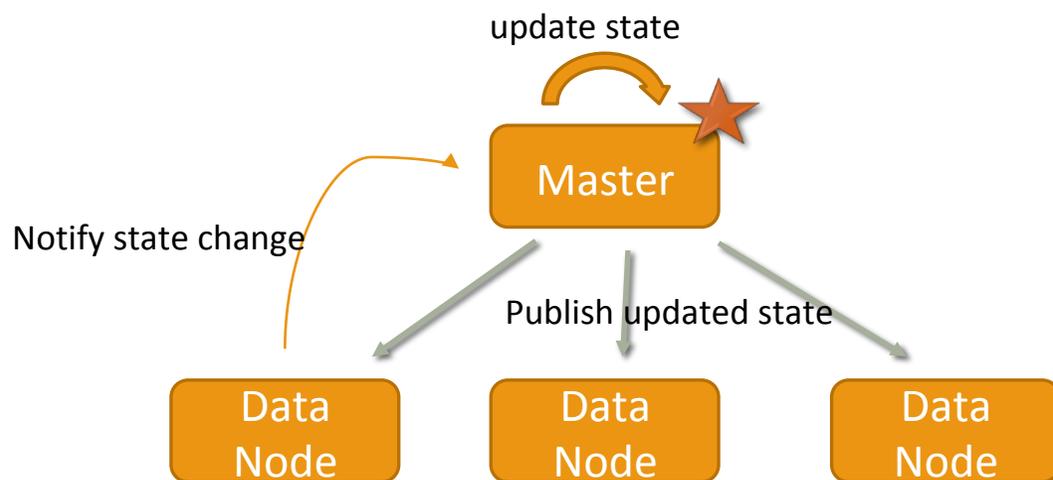
- Bigdesk, etc

选用自己顺手的监控工具，我们用Ganglia！



# 系统架构

# Master Node



Master是维护集群状态的唯一权威

集群状态是什么？

- 集群设置
- 结点成员及其角色
- 索引及其设置、mapping,etc
- shard的路由信息

状态信息不可水平扩展!!

状态信息更新开销可能很高

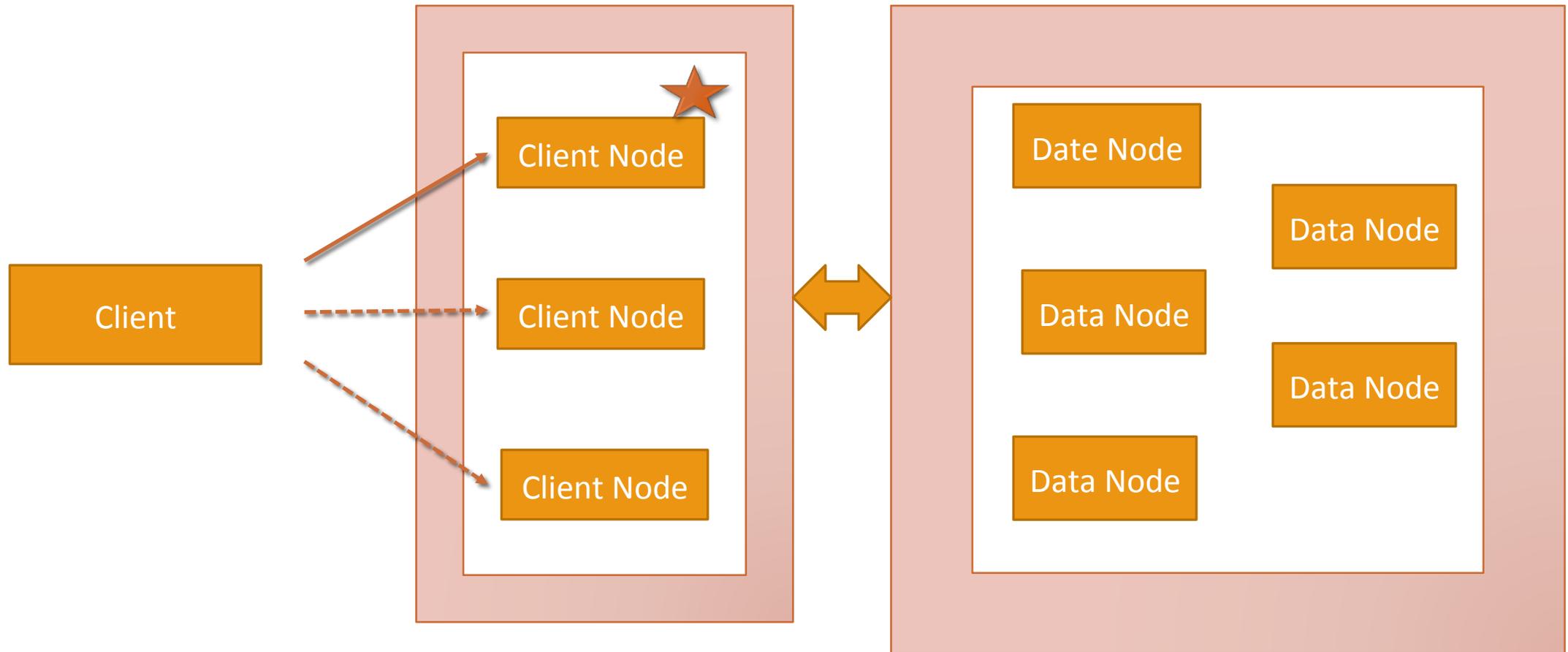
ES2.0将支持diff更新

# ES Client Node?

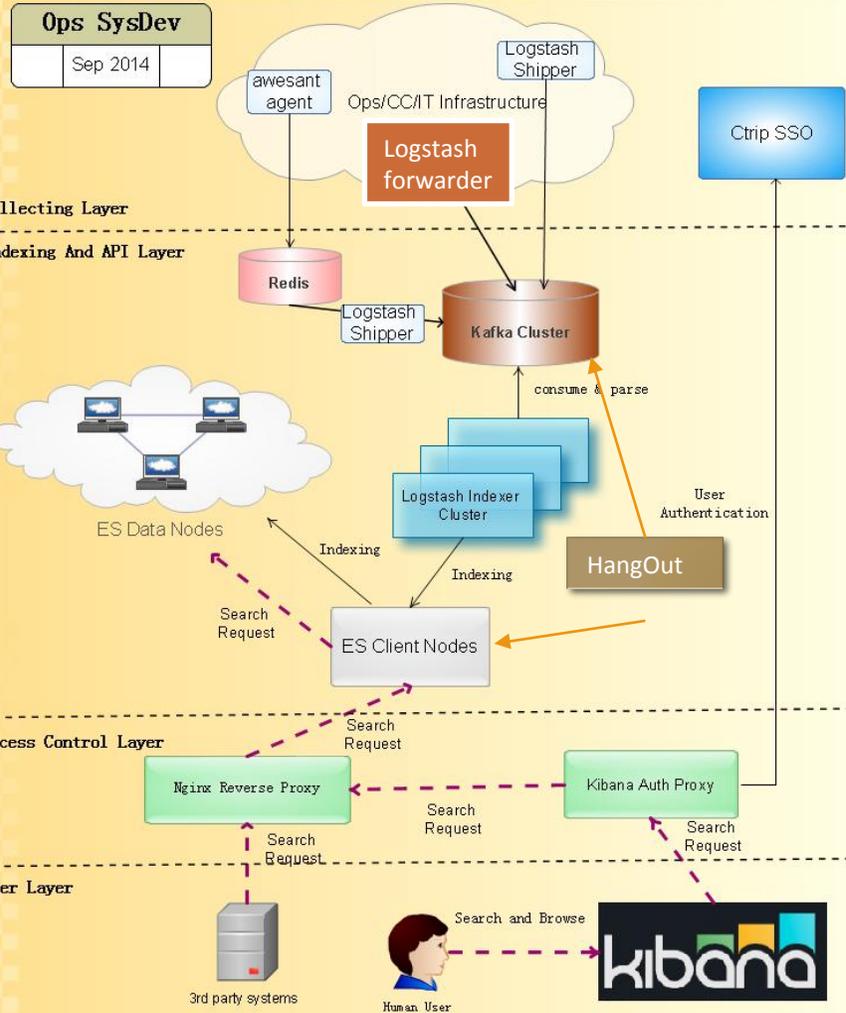
---

- 什么鬼?
  - `node.data: false`
  - 处理index和bulk request
  - 处理查询请求、负责结果汇总
  
- 为何独立出来?
  - 避免和data node争用资源
  - 故障时容易定位故障点
  - 我们生产环境client node同时也作为master node

# 独立的master/client node



# Ctrip 日志系统架构



# 日志管道 Kafka > Redis

---

- 高吞吐量，能跑满千兆网卡流量
- 磁盘队列，能容忍更长时间的消费端故障
- 集群运作，支持水平扩展和failover

# Logstash forwarder (modified)->kafka?

---

- Windows平台可选agent有限，logstash又太重
- Golang编写，可在windows上编译和运行
- 轻量级，cpu和内存消耗小
- 原生只支持lumberjack协议输出
- 开源，**经二次开发支持输出到kafka**
- 利用kafka client支持压缩的特性，节省日志传输带宽

# Kibana Auth Proxy?

---

- <https://github.com/childe/esproxy>
- 基于Django开发的kibana认证网关
- 与Ctrip SSO(基于Jasig CAS)集成，支持用户认证与单点登录
- 索引访问权限控制

写入吞吐量如何提升

# Logstash indexer

---

- grok filter很灵活但很耗cpu，尽量避免使用
- Filter worker多线程要通过命令行-w参数启用
- 输出到多个目的地的时候要小心，性能差的输出会阻塞整个通道
- ES Output plugin关键参数：
  - idle\_flush\_time # 默认1s，在可以容忍的延迟范围，尽量长
  - flush\_size # 提升到单worker吞吐量不再提升为止
  - Workers # 并非越多越好，一般1到2个足够

# HangOut

---

- <https://github.com/childe/hangout>
- 仿logstash的java实现
- 对比logstash约有5倍性能提升

# 集群状态信息与性能

---

集群状态信息过大:

- 过多的索引数量
- 过多的字段

集群状态信息频繁更新:

- 集中生成大量新索引
- 动态生成新字段



量变引起质变，集群性能及其低下

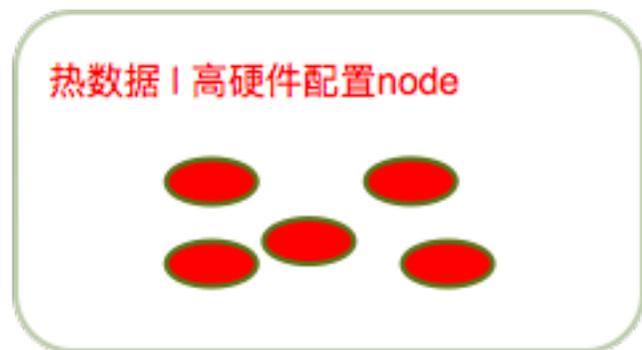
# Merge对写入速度的影响

---

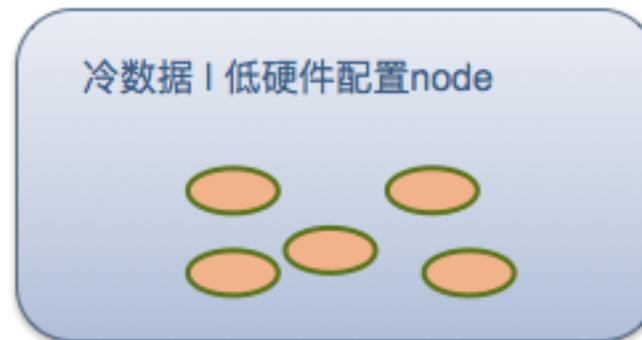
**now throttling indexing: numMergesInFlight=6, maxNumMerges=5 什么鬼?**

- Merge速率落后新segment生成速率
- ES开始限制新数据写入速率，防止segment files explosion
- 集群可能负载过高，如高开销的搜索进行中
- 也可能是集群硬件配置太好，默认的merge rate(20MB/s)设置太保守，可适当提高
- 调高索引的refresh interval和translog flush size可避免过多小segment file写到磁盘，减少小文件merge频率

# 冷热数据分离



定期自动迁移



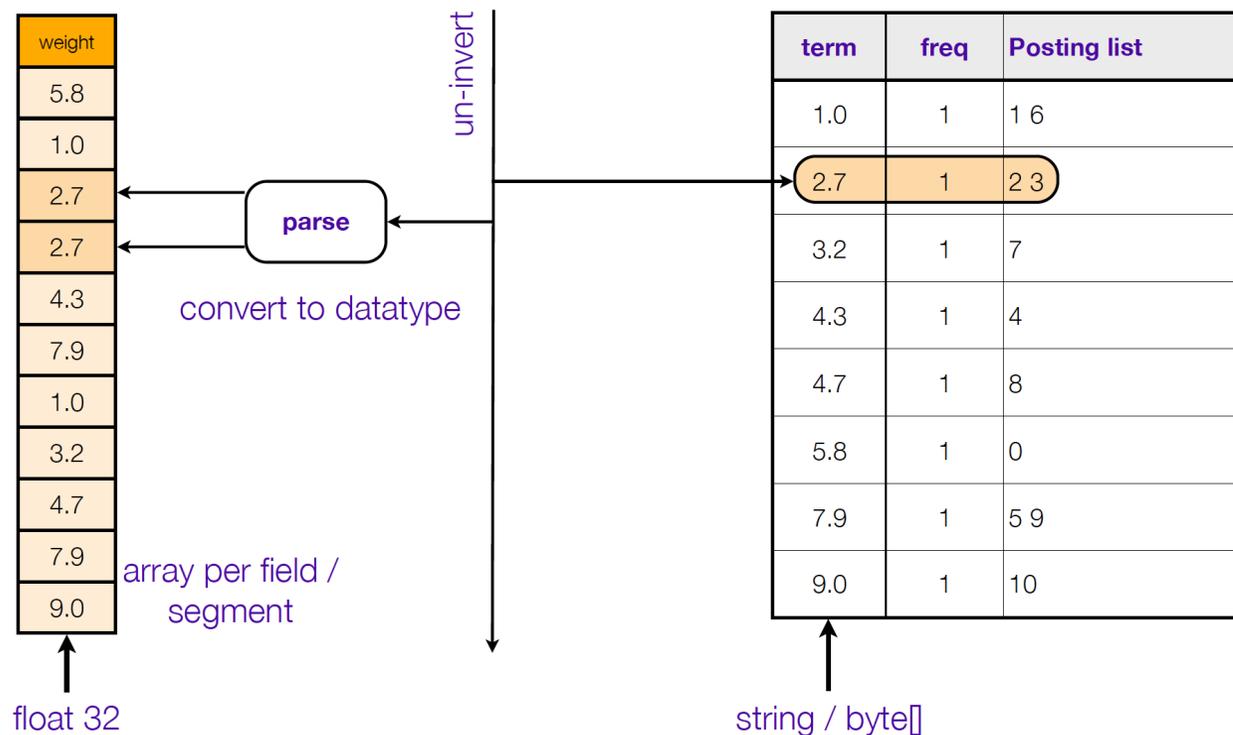
- 32core 高配机器
- 主要用于写入
- 只保留最近2天数据
- 写入速度几乎不受用户查询影响

- 8core 低配机器
- 不做写入
- 保留2天以前历史数据

优化查询、聚合速度

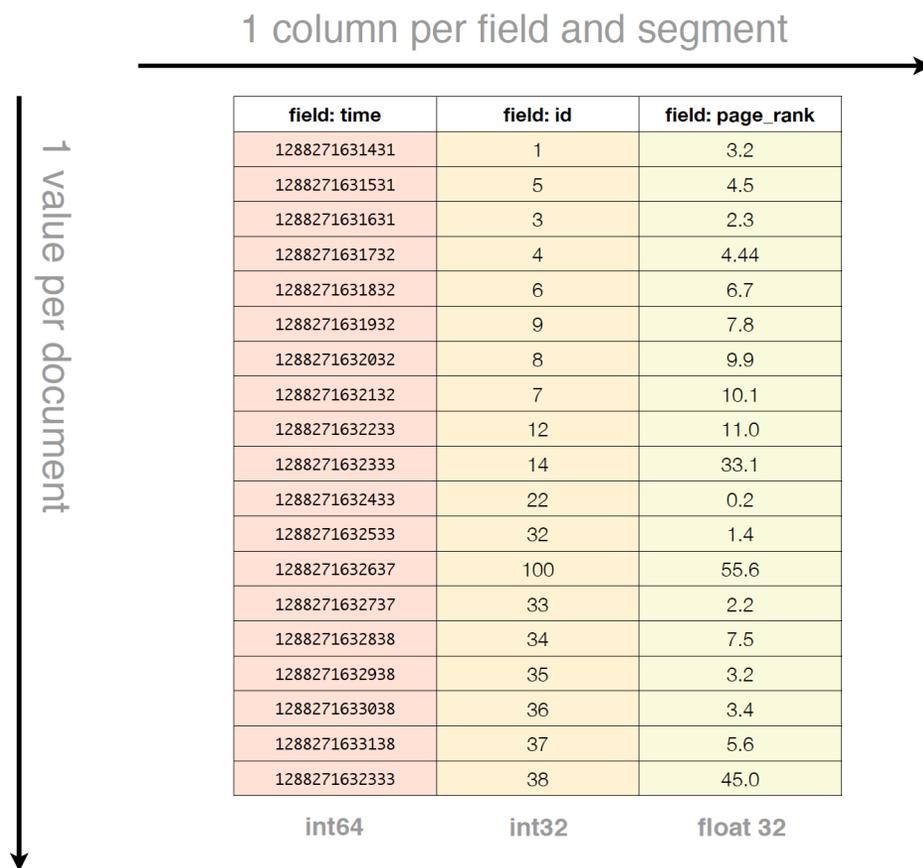
# 避免使用fielddata cache

Lucene can un-invert a field into FieldCache



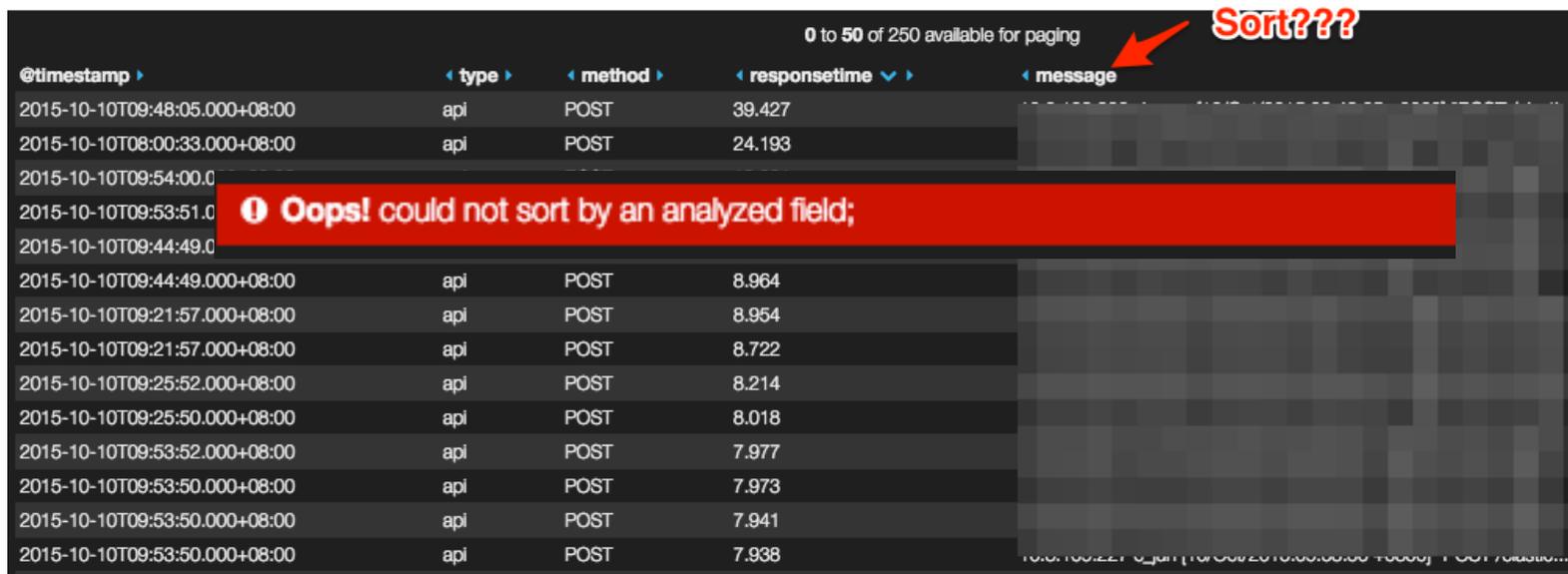
- Sort、Aggregation需要uninvert the inverted index
- 这个过程需要做解析和类型转换，开销高
- 因此结果会被cache起来加速
- Field cache会为被搜索的索引**全量**构造，而不是只为match的doc id构造
- 大索引构造fieldcache容易引起长时间Old GC，甚至OOM

# Doc Values



- Indexing时构造在磁盘上
- 顺序扫描，对文件系统cache友好
- 热数据访问等同于直接读写内存
- ES 2.0成为默认设置，之前版本需手动开启
- **Analyzed String fields不支持Doc Values**

# 避免对分词字段排序



0 to 50 of 250 available for paging

Sort???

@timestamp	type	method	responsetime	message
2015-10-10T09:48:05.000+08:00	api	POST	39.427	
2015-10-10T08:00:33.000+08:00	api	POST	24.193	
2015-10-10T09:54:00.0				
2015-10-10T09:53:51.0	Oops! could not sort by an analyzed field;			
2015-10-10T09:44:49.0				
2015-10-10T09:44:49.000+08:00	api	POST	8.964	
2015-10-10T09:21:57.000+08:00	api	POST	8.954	
2015-10-10T09:21:57.000+08:00	api	POST	8.722	
2015-10-10T09:25:52.000+08:00	api	POST	8.214	
2015-10-10T09:25:50.000+08:00	api	POST	8.018	
2015-10-10T09:53:52.000+08:00	api	POST	7.977	
2015-10-10T09:53:50.000+08:00	api	POST	7.973	
2015-10-10T09:53:50.000+08:00	api	POST	7.941	
2015-10-10T09:53:50.000+08:00	api	POST	7.938	

- Kibana对所有字段都可排序
- 但分词字段排序其实毫无意义
- 由于DocValues无法作用于分词字段，只能临时构造放到fielddata cache
- 分词字段词典一般很大
- 可能导致长时间GC甚至OOM

解决办法：

- 利用multifield，默认字段名不分词
- 教育用户
- Hack kibana，对分词字段禁用排序

# 避免deep pagination

---

GET /\_search?size=5

GET /\_search?size=5&from=5

GET /\_search?size=5&from=10

.....

GET /\_search?size=5&from=100005

- 
1. 每个shard需要返回top 100010给请求结点。
  2. 请求的结点合并所有结果，排序后返回其中5条！

# 大量拉数据

---

真实案例： 用户通过一次API调用拉取1000万条数据导致结点OOM，集群罢工

正确方法：

- Scan & scroll API分批次拉取
- 类似数据库cursor
- 数据不排序，资源消耗低

# Terms Agg – Execution Hint

The image shows a side-by-side comparison of Elasticsearch search results. On the left, the 'ordinal' mode is used, resulting in a 'took' time of 3687ms and a 'total' of 3358 hits. On the right, the 'map' mode is used, resulting in a significantly faster 'took' time of 1032ms and the same 'total' of 3358 hits. A red arrow in both screenshots points to the 'took' value. The 'map' mode is explicitly configured in the aggregation definition with the 'execution hint' set to 'map'.

```
POST /...-2015.10.09/_search?search_type=count
{
  "query": {
    "filtered": {
      "filter": {
        "bool": {
          "should": [],
          "must_not": [],
          "must": [
            {
              "range": {
                "@timestamp": {
                  "to": 1444359,
                  "from": 14443
                }
              }
            }
          ]
        }
      },
      "term": {
        "ServiceCode": 9
      }
    }
  },
  "aggs": {
    "c_ips": {
      "terms": {
        "field": "ClientID",
        "size": 50
      }
    }
  }
}
```

```
1 {
2   "took": 3687,
3   "timed_out": false,
4   "_shards": {
5     "total": 10,
6     "successful": 10,
7     "failed": 0
8   },
9   "hits": {
10    "total": 3358,
11    "max_score": 0,
12    "hits": []
13  }
14 }
```

```
POST /...-2015.10.09/_search?search_type=count
{
  "query": {
    "filtered": {
      "filter": {
        "bool": {
          "should": [],
          "must_not": [],
          "must": [
            {
              "range": {
                "@timestamp": {
                  "to": 1444359,
                  "from": 14443
                }
              }
            }
          ]
        }
      },
      "term": {
        "ServiceCode": 9
      }
    }
  },
  "aggs": {
    "c_ips": {
      "terms": {
        "field": "ClientID",
        "size": 50,
        "execution hint": "map",
      }
    }
  }
}
```

```
1 {
2   "took": 1032,
3   "timed_out": false,
4   "_shards": {
5     "total": 10,
6     "successful": 10,
7     "failed": 0
8   },
9   "hits": {
10    "total": 3358,
11    "max_score": 0,
12    "hits": []
13  }
14 }
```

```
24   "key": "...",
25   "doc_count": 23
26 },
27 {
28   "key": "...",
29   "doc_count": 21
30 },
31 {
32   "key": "...",
33   "doc_count": 21
34 },
35 }
```

- 默认terms aggs采用ordinal execution mode
- Global ordinal是lazy load的，即用的时候才去build
- 特别对于热索引，新产生segment的时候global ordinal需要rebuild
- Map直接使用field values做计算
- 如果match query的文档较少，map执行起来更快

# 索引太大搜索慢怎么办？

---

选定数据可以分片的维度，然后：

将选定维度设置为routing key，将同一个key的数据route到同一个shard

- 前提是routing key的基数(cardinality)要高，shard数量要少
- 否则可能导致数据分布不均，从而产生热点问题
- 脑洞大开的讨论：

<http://engineering.datarank.com/2015/06/30/analysis-of-hotspots-in-clusters-of-log-normally-distributed-data.html>

按照选定维度切分成不同的索引

- 切成过多的索引会导致集群状态信息过大

携程iis索引切分的故事。。。

谢谢!