日志分析场景下的搜索引擎改进

——黎吾平　日志易技术副总裁

大家好，我是黎吾平，来自日志易。日志易一直是日志产品的提供商，是 Elasticserach 的深度用户，在给用户提供服务的过程中，我们踩过很多的坑。很高兴能给与大家探讨日志分析相关的话题，今天我分享的题目是《日志处理场景下 Elasticsearch 的优化》。在今天的分享中，更多是偏向方案的解决。

开题之前，我们先看下日志处理的特点。

日志的特点

日志有以下几个特点：

1.日志是由机器产生的；

2.日志数据量大。日志本身都是由程序来产生的，产生的速度非常快，每秒几百 M 到几个 G。

3.日志是具有一定格式的数据。因此我们可以通过 ETL 来进行日志字段的提取。

4.日志是带有时间戳的。大部分时间戳出现在日志的原文中，有些在文件目录等地方，有了时间戳，我们可以基于时间范围的搜索条件去做很多的优化；

5.日志是过去的某一个事件的记录，是不可更改的，也可理解为日志的存储无更新需求。

日志搜索的特点

说完了日志处理，我们再简单说下日志搜索。

日志搜索处理大体上有四个特点：

1.新的数据是热点。实际场景中，搜索往往集中在最新的日志，历史数据搜索的概率比较小；

2.时间范围搜索。日志搜索会指定一个时间范围，这是一个很重要的特性；

3.关键字搜索，无相关性。日志搜索基本是以关键字搜索的形式进行，通常会按照时间戳或者是某一个字段进行排序；

4.统计需求多。利用日志数据进行分析，需要从多个维度进行统计。

这些日志搜索的特点，对优化都非常重要。像日志范围搜索，数据上带有时间戳信息的时候，就可以基于此做出特定的优化和过滤。

日志场景中的 Elasticsearch

下面简单的介绍一下 Elasticsearch 在日志处理中架构中的使用，及其内部相关的一些细节。

日志处理经典架构

日志处理其实有很多的解决方案。比如早期使用的脚本，后来使用的数据库，如 Hadoop/Hive，这几年则是基于 ES 的方案。基于数据库的日志处理，必须先把字段定义好。基于 ES 的方案，无需实现定义 schema，实时性会更强一点。

【P6图片】

在现在经典的日志处理架构中：

第一步是日志的搜集。日志源很多的时候遍布在用户的生产机上，搜集来的日志数据会发送到整个日志处理的集群。

第二步是日志的缓存，接收到的日志会进行缓存，通常会使用 kafka 多一些。

第三步会做一些数据的预处理，比如字段的抽取，实时的日志串联等等。

第四步是索引和存储，这一部分就是 ES 承担的角色。这部分需要将接收来的数据建立索引。

第五步是数据分析，主要是做一些搜索和统计。这需要基于ES或其他服务提供的接口，做一些可视化展示等，以满足具体的分析需求。

建立索引和搜索，对 ES 有很高的要求，需要满足高性能，高可用。如果性能达不到要求，可用性就会受到影响。

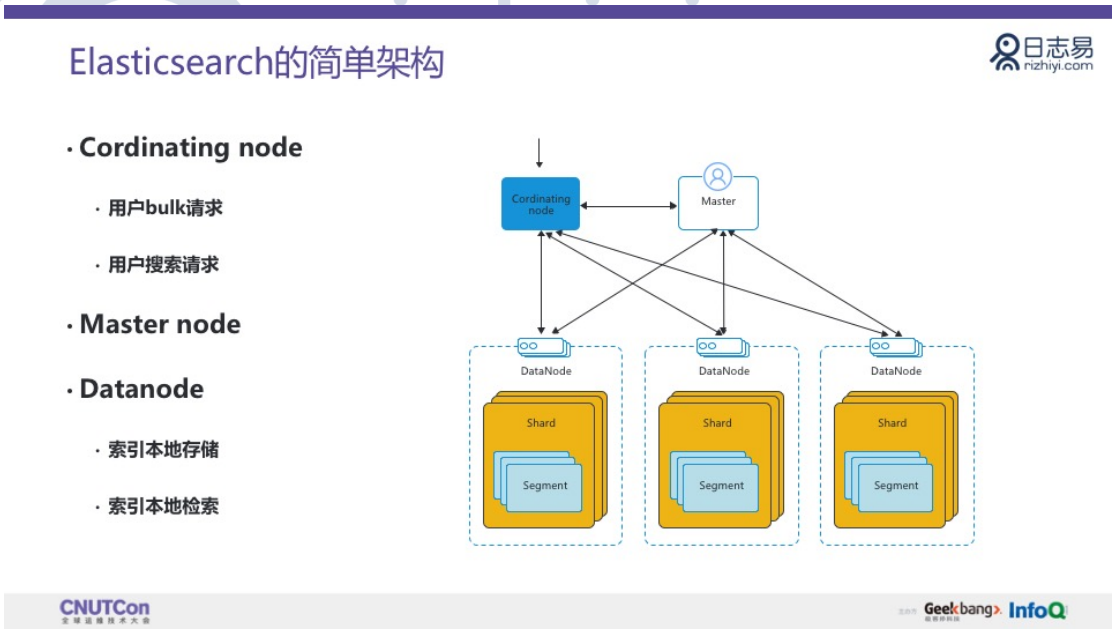
ES的架构

了解了日志处理的流程架构，下面简单的给大家介绍一下 ES 的架构。这里只画出了本次分享中需要涉及三个类型的节点。

第一个是 Cordinating node（节点）。负责转发用户的 bulk 或者 search 请求给 shard，合并 shard 返回的请求，并返回给用户。

第二个 Master node，Master node 承担集群的管理功能，如集群节点的管理，索引的管理等等。

第三个是Data node，Data node 实际对数据进行本地索引，以及执行本地的搜索。



【P7-图片】

ES在实际场景中的问题

在实际的使用场景中，我们会碰到 ES 的很多的问题。

第一个是字段类型的不兼容。对 ES 来说，如果字段不兼容，将无法索引。

第二个问题是索引过程性能消耗太大。用户给日志处理的资源本就不多，索引的过程就会消耗50%以上的资源，这样留给搜索的资源就会更少。

第三个是实时性达不到要求。准实时是一个分钟级别的东西，但是很多的用户会提出五秒甚至是一二秒的要求。对此 ES 也可以做，但是代价是非常大的。

第四个是索引太多，这会影响稳定性和性能。实际生产中，由于用户需求等原因，会把数据拆分成很多索引。

第五个是搜索请求可能导致集群响应慢或者是 OOM 等。一个大的搜索请求可能会命中很多的 doc，这会导致消耗的资源很大，继而对其他并发的搜索请求影响很大，甚至导致 OOM。

最后一个问题是 GC 的问题。大的搜索申请的资源非常的大，会导致索引掉线。

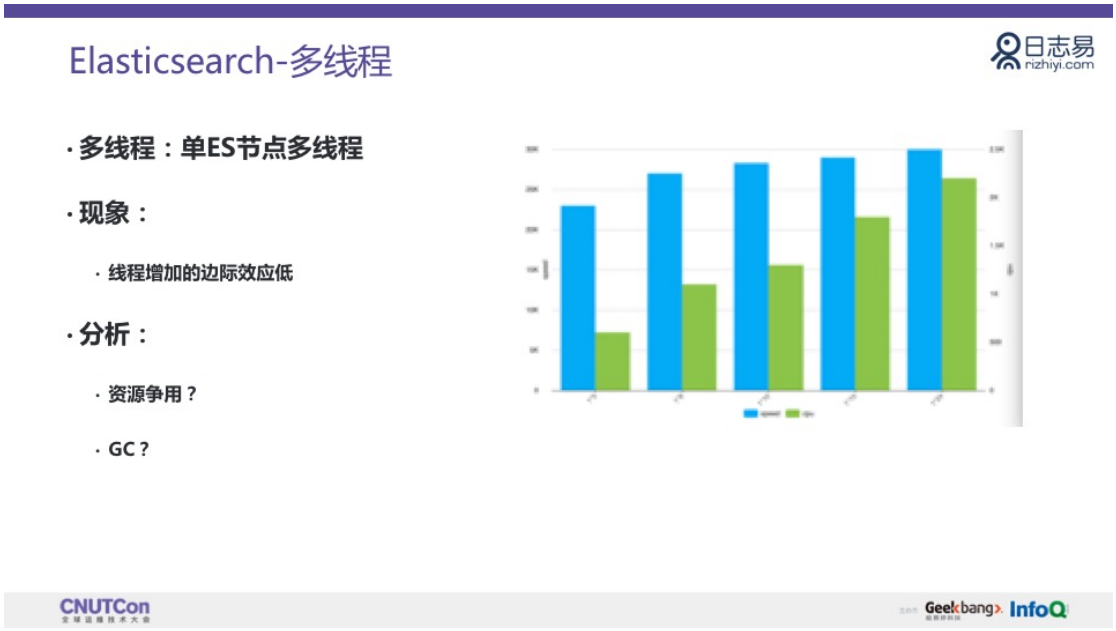
Elasticsearch 优化方案

以上主要说了日志处理的特点和实际使用中ES的特点。接下来就和大家探讨下 ES 的优化方案，以及日志易是针对 ES 的哪些问题做出优化的。

ES 是一个通用的搜索引擎，而日志处理是有特殊的日志需求的。由此进行的优化，方向主要集中在几个步骤：第一个是入库优化，从给搜索留下更多的资源上着手；第二个是搜索性能的优化；第三个是稳定性优化，最后是功能增加。

-入库优化

先看入库优化，在 ES 单节点多线程的场景中，可以看到线程和 CPU 的增长并不是等比例的。下图中左边是入库的速度，右边的单位是 CPU 的消耗，可以看出线程和 CPU 的增长并不是等比例的。很多银行都会使用高配的机器，这种部署的方式就很不经济。一个优化的方法是，单机上部署多个 ES 节点，每个节点的内存会更小，线程数也较少。这种部署方式会较大幅度的提高入库性能。



- Mapping 更新慢的优化

第 2 个问题是 Mapping 的更新慢。在实际的场景中需要创建大量的索引，比如说时序数据按时间切分索引，这样可以针对冷热数据做出不同的处理，数据的流动就会很方便。再比如说将不同的业务划分不同的索引，这样搜索可以缩小到更小的范围内。

在新索引创建的时候，Mapping 是空的，因为不允许字段出现不兼容的情况，所以新索引创建会做全局检测。全局检测需要通过 Master 执行，所有 Data node 在碰到新的字段时，通过 Master 进行全局冲突检测，在集群大的时候 Master 压力很大，在一些用户那里，新索引创建时会堵一个半小时无法入库，那一个半小时的搜索数据是不对的。

日志易自研引擎做出优化之后，Mapping 更新慢的问题得到了有效解决。

-时间范围搜索优化

第3个是 Elasticsearch 时间范围搜索。搜索的时候会有很多的时序日志，一般会指定一个时间范围。对于索引，可以在索引上加上时间范围信息，可以先根据时间范围过滤掉少量需要搜索的索引， 对于索引内部的 Segment，ES 并未保存时间戳，不知道时间范围，带来的情况是必须去搜索所有的 Segment。

通过更加精细的时间戳 Meta 信息，可以充分利用时间范围搜索的优势，过滤掉无锁搜索的索引或者 Segment，优化效果还是很明显的。

-索引 Meta 加载的优化

然后是索引 Meta 的加载。ES中，fdx、tvx、fnm、si、tip等文件是常驻内存的，虽然

搜索时不用加载这些文件，但实际场景中，ES加载的大量文件会导致打开索引很慢，因为加载了太多的东西占用了相当大的内存。如果出于性能考虑，把索引关闭的话，当有节点出现故障的时候，数据就丢失了。

前面提到最新的数据是热点，实际上可以理解为最新的索引是搜索热点，索引中可能一部分数据是热点，因此可以通过 Cache 的方式来进行 Meta 的加载，减小索引打开消耗的内存。

-其他优化

还有一些优化点，我们就不再展开说了，比如 doc 去重逻辑优化，Aggregation 的内存控制优化等等。

日志易自研日志搜索引擎

前面主要是说在 ES 的代码基础上做出的优化。

ES 的定位是一个通用的系统。ES 虽然有 50% 的场景是做日志处理，但是 ES 本身还是会处理很多其他的任务，很多公司将 ES 用在搜索上。日志易是专门做日志处理的，在这点上就可以有很多的优化策略。

ES 的代码量真的很大，将近有 40 多万的代码，每次改动都是心惊胆战的，测试的压力也很大。

而且 ES 是开源软件，维护成本很大。

基于以上若干原因，日志易自己做了一套搜索引擎，内部叫 Beaver。这个自研引擎是用 C++ 开发的，主要是速度更快，内存好控制。

自研引擎的实时性改进

日志易如何解决实时性的问题的呢？首先我们知道，ES 是准实时的系统，在入库的时候，日志发送到 shard 上进行索引，首先在内存中构建 Segment，然后通过 refresh 成可搜索的 Segment。实时性要求越高，Refresh 时间间隔就需要越小，则 Segment 越小，搜索越慢。为了避免大量小的 Segment， Lucene 按照一定策略对 Segment 进行 Merge，Merge 对资源的消耗还是很大的。

Beaver 引擎对以上做出了优化，建立可搜索的内存索引，避免频繁的 Refresh，既提高了实时性，又可以降低 Merge 的代价。

自研引擎的 Replica 策略改进

ES 通过 Replica 来保证数据可靠性，每个 Shard 一定有一个 Primary，0 个或者多个 Replica，Primary 和 Replica 都可读可写，Replica 和 Primary 执行同样的索引过程，有一些重复的资源消耗。

我们对 Replica 策略进行调整，改进后大大减小了因副本导致的资源消耗，同时保证了数据的可靠性。

自研引擎的索引分层改进

关于索引分层，ES 的方案是冷热机器分离，缺点是跨机器拷贝索引，网络压力大。日志易也支持冷热分离，冷热索引使用不同的路径，冷数据由于使用较少，可以删除部分索引中的数据，降低索引的膨胀率当作日志条目的备份。

自研引擎的 Merge 优化

关于搜索结果 Merge 优化，简单来说，ES 搜索结果的 Merge，由 Cordinating node 合并，本次搜索命中的 Shard 返回结果执行一次合并，这样可能造成单 Merge 节点 CPU 消耗过大，或者集群过大时内存消耗极大，出现 OOM。ES 搜索中，单个 Merge 可能成为集群扩展的瓶颈。

日志易自研引擎在搜索结果的进行分层 Merge，降低单节点的压力，以及 OOM 的可能性，集群的扩展性得到有效提升。

日志易自研引擎的其他优化

最后说下日志易自研 Beaver 引擎其他的优化点。Global ordinal 优化、搜索时抽取字段、更好的任务管理、更严格的内存控制等等。

日志易自研引擎的优化效果

最后是说的优化的效果，日志易自研 Beaver 引擎入库的性能提升了大约 500%，搜索性能提升了 200%，稳定性 OOM 的情况基本不会再出现，节点掉线的情况减少。

我的内容大概就这么多。