链滴

Elasticsearch 常见面试题

作者：Anileh

原文链接：https://ld246.com/article/1665318821692

来源网站：链滴

<h2 id="为什么要使用-Elasticsearch">为什么要使用 Elasticsearch</h2>
系统中的数据， 随着业务的发展，时间的推移， 将会非常多， 而业务中往往采用模糊查询进行据的搜索， 而模糊查询会导致查询引擎放弃索引，而去进行全表扫描。在百万级别的数据库中，查询率将非常低下。
此时使用 ES 做一个全文索引，将经常查询的某些内容，比如说电商系统的商品表中商品名，描述、格等这些字段放入 ES 索引库中，可以提高查询速度。
<h2 id="ES-的-master-选举流程">ES 的 master 选举流程</h2>

该过程是 ZenDiscovery 模块负责的，主要包含 Ping （节点之间通过此 RPC 来发现彼此）和 Uncast（单模块包含一个主机列表以控制要 Ping 的节点）
对所有可以成为 master 的节点（node master: true）根据 nodeId 字典排序，每次选举各个节都把自己所知道节点排一次序，然后选出第一个（第 0 位）节点，暂且认为它是 master
如果某个节点的投票数达到一定值（可以成为 master 节点数 n/2+1），并且该节点也选举自己 那这个节点就是 master。否则重新选举一直到满足上述条件

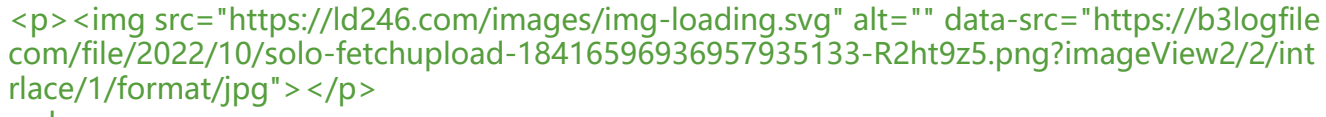
master 节点的职责主要包括集群、节点和索引的管理，不负责文档级别的管理；
data 节点可以关闭 http 功能。
<h2 id="Elasticsearch-集群脑裂问题">Elasticsearch 集群脑裂问题</h2>
“脑裂”问题的可能成因：

网络问题：集群间的网络延迟导致一些节点访问不到 master， 误认为 master 挂掉了从而选举新的 master，并对 master 上的分片和副本标红，分配新的主分片。
节点负载：主节点既是 master 还是 data，访问量过大 导致 ES 停止响应造成大面积延迟，此时他节点认为主节点已宕机，会重新选取主节点。
内存回收：data 节点上的 ES 进程占用的内存较大，引发 JVM 的大规模内存回收，使得 ES 进失去响应。

脑裂问题解决方案：

减少误判：<code>discovery.zen.ping_timeout</code> 节点状态的响应时间默认 3s，可以当调大，如果 master 在该响应时间内没有做出响应应答，判断该节点已经挂掉了。
调大参数（如 6s，discovery.zen.ping_timeout:6），可适当减少误判。
选举触发：<code>discovery.zen_minimum._master_nodes: 1</code>，该参數用于控制选行为的最小集群主节点数量。当备选主节点的个數大于等于该参数的值，且备选主节点中有该参数个点认为主节点挂了，进行选举。官方建议为(n / 2) +1, n 为主节点个数（即有资格成为主节点的节点数）。
角色分离：master 节点与 data 节点分离

主节点配置为：node master: true，node data: false
从节点配置为：node master: false，node data: true

<h2 id="ES索引文档的流程">ES 索引文档的流程</h2>


协调节点默认使用文档 ID 参与计算（也支持 routing），为路由提供合适的分片：<code>shard = hash(document_id) % (num_of_primary_shards)</code>
当分片所在节点接收到来自协调节点的请求后，将请求写入到 Memory Buffer，然后定时（默间隔 1s）写入到 Filesystem Cache，这个从 Memory Buffer 到 Filesystem Cache 的过程称为 refresh
在某些情况下，存在 Memoery Buffer 和 Filesystem Cache 数据丢失， ES 通过 Translog 机

保证数据的可靠性：接收到请求后，同时写入到 Translog 中，当 Filesystem Cache 中的数据写入磁盘中时，才会清除掉 Translog，这个过程叫做 flush。
在 flush 过程中，内存中的缓冲将被清除，内容被写入一个新段，段的 fsync 将创建一个新的提点，并将内容刷新到磁盘，旧的 Translog 将被删除并开始新的 Translog
flush 触发时机：定时触发（默认 30 分钟）或者 Translog 变得太大（默认为 512M）

<h2 id="ES-更新和删除文档的流程">ES 更新和删除文档的流程</h2>

删除和更新都是写操作，但是 ES 中的文档是不可变的，因此不能被删除或者改动
磁盘上的每个段都有一个相应的 .del 文件。当删除请求发送后，文档并没有真的被删除，而是<b
>
在 .del 文件中被标记为删除。该文档依然能匹配查询，但是会在结果中被过滤掉。当段合并
时，在 .del 文件中被标记为删除的文档将不会被写入新段。
在新的文档被创建时， Elasticsearch 会为该文档指定一个版本号。当执行更新时，旧版本的文在.del 文件中被标记为删除，新版本的文档被索引到一个新段。旧版本的文档依然能匹配查询，但是在结果中被过滤掉。

<h2 id="ES-的搜索流程">ES 的搜索流程</h2>
<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/file/2022/10/solo-fetchupload-7251036235952805104-2mxPHB9.png?imageView2/2/int
rlace/1/format/jpg">

搜索被执行成一个两阶段过程，我们称之为 Query Then Fetch；
在初始查询阶段时，查询会广播到索引中每一个分片拷贝（主分片或者副本分片）。 每个分片<b
>
在本地执行搜索并构建一个匹配文档的大小为 from + size 的优先队列。 PS：在搜索的时候是会查询
ilesystem Cache 的，但是有部分数据还在 Memory Buffer，所以搜索是近实时的。
每个分片返回各自优先队列中 所有文档的 ID 和排序值 给协调节点，它合并这些值到自己的优先列中来产生一个全局排序后的结果列表。
取回阶段， 协调节点辨别出哪些文档需要被取回并向相关的分片提交多个 GET 请求。每个分<br

片加载并丰富文档，如果有需要的话，接着返回文档给协调节点。一旦所有的文档都被取回了，协调点返回结果给客户端。
Query Then Fetch 的搜索类型在文档相关性打分的时候参考的是本分片的数据，这样在文档<br

数量较少的时候可能不够准确， DFS Query Then Fetch 增加了一个预查询的处理，询问 Term 和 D
cument frequency，这个评分更准确，但是性能会变差。

<h2 id="Elasticsearch-在部署时-对-Linux-的设置有哪些优化方法">Elasticsearch 在部署时，对 Li
ux 的设置有哪些优化方法</h2>

尽可能大的机器内存： 64 GB 内存的机器是非常理想的， 不要少于 8 GB
更快的 CPU 和更多的核心之间选择：更多的核心更好，多个内核提供的额外并发远胜过稍微快点点的时钟频率。
SSD：基于 SSD 的节点，查询和索引性能都有提升。
避免集群跨越多个数据中心：绝对要避免集群跨越大的地理距离。
确保应用程序的 JVM 和服务器的 JVM 是完全一致
设置 gateway.recover_after_nodes、 gateway.expected_nodes、 gateway.recover_after_ti
e ： 在集群重启的时候避免过多的分片交换，加速数据恢复
使用单播代替组播：Elasticsearch 默认被配置为使用单播发现，以防止节点无意中加入集群。只在同一台机器上运行的节点才会自动组成集群
不要随意修改垃圾回收器（CMS）和各个线程池的大小
把内存的（少于）一半给 Lucene（但不要超过 32 GB！），通过 ES_HEAP_SIZE 设置。
增加文件描述符：Lucene 使用了大量的文件， Elasticsearch 在节点和 HTTP 客户端之间的通

也使用了大量的 scoket。 所有这一切都需要足够的文件描述符。

<h2 id="GC方面的注意点">GC 方面的注意点</h2>

倒排词典的索引需要常驻内存，无法 GC，需要监控 data node 上 segment memory 增长趋势
各类缓存， field cache, filter cache, indexing cache, bulk queue 等等，要设置合理的大小，且要应该根据最坏的情况来看 heap 是否够用，也就是各类缓存全部占满的时候，还有 heap 空间可分配给其他任务吗？避免采用 clear cache 等“自欺欺人”的方式来释放内存。
避免返回大量结果集的搜索与聚合。确实需要大量拉取数据的场景，可以采用 scan & scroll api 来实现。
cluster stats 驻留内存并无法水平扩展，超大规模集群可以考虑分拆成多个集群通过 tribe node 连接。
想知道 heap 够不够，必须结合实际应用场景，并对集群的 heap 使用情况做持续的监控。

<h2 id="ES-聚合大数据量的实现机制">ES 聚合大数据量的实现机制</h2>
Elasticsearch 提供的首个近似聚合是 cardinality 度量。它提供一个字段的基数，即该字段的 distnct 或者 unique 值的数目，基于 HLL 算法。
HLL 会先对我们的输入作哈希运算，然后根据哈希运算的结果中的 bits 做概率估算从而得到基数。特点是：

可配置的精度，用来控制内存的使用（更精确 = 更多内存）
小的数据集精度是非常高的
我们可以通过配置参数，来设置去重需要的固定内存使用量
无论数千还是数十亿的唯一值，内存使用量只与配置的精确度相关

<h2 id="ES-高并发">ES 高并发</h2>

乐观并发控制，使用版本号确保新版本不会被旧版本覆盖，由应用层来处理具体的冲突；
对于写操作，一致性级别支持 quorum/one/all，默认为 quorum，即只有当大多数分片可用
时才允许写操作。但即使大多数可用，也可能存在因为网络等原因导致写入副本失败，这样该副本被为故障，分片将会在一个不同的节点上重建。
对于读操作，当 replication 为 sync(默认)时，操作在主分片和副本分片都完成后才会返回；当 rplication 为 async 时 ，也可以通过设置搜索请求参数_preference 为 primary 来查询主分片，确保档是最新版本。

<h2 id="如何监控-ES-集群状态">如何监控 ES 集群状态</h2>

elasticsearch-head 插件。
通过 Kibana 监控 Elasticsearch。你可以实时查看你的集群健康状态和性能，也可以分析过去的群、索引和节点指标

<h2 id="字典树">字典树</h2>
字典树又称单词查找树，Trie 树，是一种树形结构，是一种哈希树的变种。典型应用是用于统计排序和保存大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的点是：利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希高。
Trie 的核心思想是空间换时间，利用字符串的公共前缀来降低查询时间的开销以达到提高效率的的。它有 3 个基本性质：

根节点不包含字符，除根节点外每一个节点都只包含一个字符。
从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。
每个节点的所有子节点包含的字符都不相同。

对于中文的字典树，每个节点的子节点用一个哈希表存储，这样就不用浪费太大的空间，而且查速度上可以保留哈希的复杂度 O(1)。
<h2 id="ES-中的集群-节点-索引-文档-类型是什么">ES 中的集群、节点、索引、文档、类型是什么/h2>

集群是一个或多个节点（服务器）的集合，它们共同保存数据，并提供跨所有节点的联合索引和索功能。集群由唯一名称标识，默认为"elasticsearch"。因为如果节点设置为按名称加入群集，则该点只能是群集的一部分。
节点是属于集群一部分的单个服务器。它存储数据并参与群集索引和搜索功能。索引就像关系数库中的“数据库”。它有一个定义多种类型的映射。索引是逻辑名称空间，映射到一个或多个主分片并且可以有零个或多个副本分片。
MySQL => 数据库，Elasticsearch => 索引。
文档类似于关系数据库中的一行。不同之处在于索引中的每个文档可以具有不同的结构(字段)，是对于通用字段应该具有相同的数据类型。
MySQL => Databases => Tables => Columns / Rows，Elasticsearch=> Indices =gt; Types => 具有属性的文档 Doc。
类型是索引的逻辑类别/分区，其语义完全取决于用户。

<h2 id="ES-中的倒排索引是什么-">ES 中的倒排索引是什么、</h2>
倒排索引是搜索引擎的核心。搜索引擎的主要目标是在查找发生搜索条件的文档时提供快速搜索
ES 中的倒排索引其实就是 Lucene 的倒排索引， 倒排索引会在存储数据时将关键词和数据进行关联保存到倒排表中。
当查询时，将查询内容进行分词后在倒排表中进行查询，最后匹配数据即可。