

ElasticSearch 实战之千万级 TPS 写入

作者: [fc13240](#)

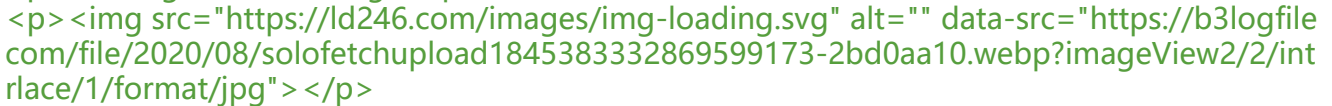
原文链接: <https://ld246.com/article/1597397668476>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

***1. ***

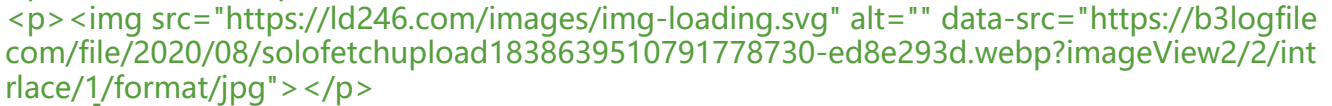
背景



前段时间，为了降低用户使用 Elasticsearch 的存储成本，我们做了数据的冷热分离。为了保持群磁盘利用率不变，我们减少了热节点数量。ElasticSearch 集群开始出现写入瓶颈，节点产生大量写入 rejected，大量从 kafka 同步的数据出现写入延迟。我们深入分析写入瓶颈，找到了突破点，最终将 Elasticsearch 的写入性能提升一倍以上，解决了 Elasticsearch 瓶颈导致的写入延迟。这篇文章介绍了我们是如何发现写入瓶颈，并对瓶颈进行深入分析，最终进行了创新性优化，极大的提升了写入能。

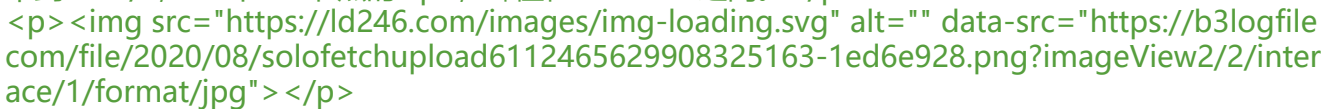
***2. ***

写入瓶颈分析

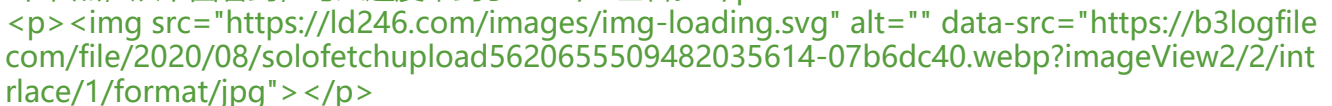


2.1 发现瓶颈

我们去分析这些延迟问题的时候，发现了一些不太好解释的现象。之前做性能测试时，ES 节点 cpu 利用率能超过 80%，而生产环境延迟索引所在的节点 cpu 资源只使用了不到 50%，集群平均 cpu 利用率不到 40%，这时候 IO 和网络带宽也没有压力。通过提升写入资源，写入速度基本没增加。于是我们开始一探究竟，我们选取了一个索引进行验证，该索引使用 10 个 ES 节点。从下图看到，写入速不到 20w/s，10 个 ES 节点的 cpu，峰值在 40-50% 之间。



为了确认客户端资源是足够的，在客户端不做任何调整的情况下，将索引从 10 个节点，扩容到 1 个节点，从下图看到，写入速度来到了 30w/s 左右。



这证明了瓶颈出在服务端，ES 节点扩容后，性能提升，说明 10 个节点写入已经达到瓶颈。但是图可以看到，CPU 最多只到了 50%，而且此时 IO 也没达到瓶颈。

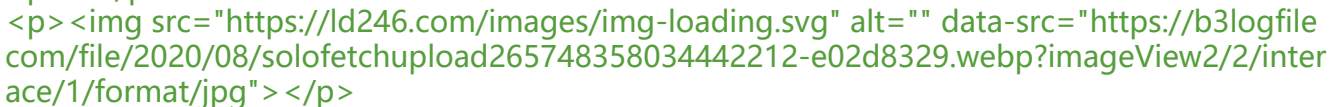
2.2 ES 写入模型说明

这里要先对 ES 写入模型进行说明，下面分析原因会跟写入模型有关。

这里要先对 ES 写入模型进行说明，下面分析原因会跟写入模型有关。

这里要先对 ES 写入模型进行说明，下面分析原因会跟写入模型有关。

这里要先对 ES 写入模型进行说明，下面分析原因会跟写入模型有关。



客户端一般是准备好一批数据写入 ES，这样能极大减少写入请求的网络交互，使用的是 ES 的 Bulk 接口，请求名为 BulkRequest。这样一批数据写入 ES 的 ClientNode。ClientNode 对这一批数据按数据中的 routing 值进行分发，组装成一批 BulkShardRequest 请求，发送给每个 shard 所在的 DataNode。发送 BulkShardRequest 请求是异步的，但是 BulkRequest 请求需要等待全部 BulkShardRequest 响应后，再返回客户端。

2.3 寻找原因

我们在 ES ClientNode 上有记录 BulkRequest 写入 slowlog。

- `items` 是一个 BulkRequest 的发送请求数
- `totalMills` 是 BulkRequest 请求的耗时
- `max` 记录的是耗时最长的 BulkShardRequest 请求
- `avg` 记录的是所有 BulkShardRequest 请求的平均耗时。

<p>我这里截取了部分示例。 </p>

<p>| [xxx][INFO][o.e.m.r.RequestTracker] [log6-clientnode-sf-5aaae-10] bulkDetail||requestId=null||size=10486923||items=7014||totalMills=2206||max=2203||avg=37[xxx][INFO][o.e.m.r.RequestTracker] [log6-clientnode-sf-5aaae-10] bulkDetail||requestId=null||size=210506||item=137||totalMills=2655||max=2655||avg=218 |</p>

<p>从示例中可以看到，2 条记录的 avg 相比 max 都小了很多。一个 BulkRequest 请求的耗时，决于最后一个 BulkShardRequest 请求的返回。这就很容易联想到分布式系统的长尾效应。 </p>

<p> </p>

<p>接下来再看一个现象，我们分析了某个节点的 write 线程的状态，发现节点有时候 write 线程全是 runnable 状态，有时候又有大量在 waiting。此时写入是有瓶颈的，runnable 状态可以理解，但却常出现 waiting 状态。所以这也能印证了 CPU 利用率不高。同时也论证长尾效应的存在，因为长尾节点繁忙，ClientNode 在等待繁忙节点返回 BulkShardRequest 请求，其他节点可能出现相对空闲的状态。下面是一个节点 2 个时刻的线程状态： </p>

<p>时刻一： </p>

<p> </p>

<p>时刻二： </p>

<p> </p>

<p>****| 2.4 瓶颈分析 </p>

<p>谷歌大神 Jeffrey Dean 《The Tail At Scale》介绍了长尾效应，以及导致长尾效应的原因。总结起来，就是正常请求都很快，但是偶尔单次请求会特别慢。这样在分布式操作时会导致长尾效应。我们从 ES 原理和实现中分析，造成 ES 单次请求特别慢的原因。发现了下面几个因素会造成长尾问题： </p>

<p>2.4.1 lucene refresh </p>

<p>**

**</p>

<p>我们打开 lucene 引擎内部的一些日志，可以看到： </p>

<p> </p>

<p>write 线程是用来处理 BulkShardRequest 请求的，但是从截图的日志可以看到，write 线程也会进行 refresh 操作。这里面的实现比较复杂，简单说，就是 ES 定期会将写入 buffer 的数据 refresh 成 segment，ES 为了防止 refresh 不过来，会在 BulkShardRequest 请求的时候，判断当前 shard 否有正在 refresh 的任务，有的话，就会帮忙一起分摊 refresh 压力，这个是在 write 线程中进行的这样的问题就是造成单次 BulkShardRequest 请求写入很慢。还导致长时间占用了 write 线程。在 write queue 的原因会具体介绍这种危害。 </p>

<p> </p>

<p>**2.4.2 **translog ReadWriteLock </p>

<p>**

**</p>

<p>ES 的 translog 类似 LSM-Tree 的 WAL log。ES 实时写入的数据都在 lucene 内存 buffer 中，以需要依赖写入 translog 保证数据的可靠性。ES translog 具体实现中，在写 translog 的时候会上 ReadLock。在 translog 过期、翻滚的时候会上 WriteLock。这会出现，在 WriteLock 期间，实时写会等待 ReadLock，造成了 BulkShardRequest 请求写入变慢。我们配置的 translog 写入模式是 asyn，正常开销是非常小的，但是从图中可以看到，写 translog 偶尔可能超过 100ms。 </p>

<p> </p>

ace/1/format/jpg"></p>

<p>****2.4.3 ****write queue</p>

<p>**

**</p>

<p></p>

<p>ES DataNode 的写入是用标准的线程池模型是，提供一批 active 线程，我们一般配置为跟 cpu 个数相同。然后会有一个 write queue，我们配置为 1000。DataNode 接收 BulkShardRequest 请求，先将请求放入 write queue，然后 active 线程有空隙的，就会从 queue 中获取 BulkShardRequest 请求。这种模型下，当写入 active 线程繁忙的时候，queue 中会堆积大量的请求。这些请求在等待行，而从 ClientNode 角度看，就是 BulkShardRequest 请求的耗时变长了。下面日志记录了 action 的 slowlog，其中 waitTime 就是请求等待执行的时间，可以看到等待时间超过了 200ms。</p>

<p>| [xxx][INFO][o.e.m.r.RequestTracker] [log6-datanode-sf-4f136-100] actionStats|action indices:data/write/bulk[s][p]||requestId=546174589||taskId=6798617657||waitTime=231||total ime=538| [xxx][INFO][o.e.m.r.RequestTracker] [log6-datanode-sf-4f136-100] actionStats|action indices:data/write/bulk[s][p]||requestId=546174667||taskId=6949350415||waitTime=231||totalTime=548 |</p>

<p>*****2.4.4 **JVM GC</p>

<p>ES 正常一次写入请求基本在亚毫秒级别，但是 jvm 的 gc 可能在几十到上百毫秒，这也增加了 BulkShardRequest 请求的耗时。这些加重长尾现象的 case，会导致一个情况就是，有的节点很繁忙，往这个节点的请求都 delay 了，而其他节点却空闲下来，这样整体 cpu 就无法充分利用起来。</p>

<p>**| **2.5 论证结论</p>

<p>长尾问题主要来自于 BulkRequest 的一批请求会分散写入多个 shard，其中有的 shard 的请求因为上述的一些原因导致响应变慢，造成了长尾。如果每次 BulkRequest 只写入一个 shard，那么不存在写入等待的情况，这个 shard 返回后，ClientNode 就能将结果返回给客户端，那么就不存在尾问题了。</p>

<p>我们做了一个验证，修改客户端 SDK，在每批 BulkRequest 写入的时候，都传入相同的 routing 值，然后写入相同的索引，这样就保证了 BulkRequest 的一批数据，都写入一个 shard 中。</p>

<p></p>

<p>优化后，第一个平稳曲线是，每个 bulkRequest 为 10M 的情况，写入速度在 56w/s 左右。之将 bulkRequest 改为 1M (10M 差不多有 4000 条记录，之前写 150 个 shard，所以 bulkSize 比大) 后，性能还有进一步提升，达到了 65w/s。</p>

<p>从验证结果可以看到，每个 bulkRequest 只写一个 shard 的话，性能有很大的提升，同时 cpu 能充分利用起来，这符合之前单节点压测的 cpu 利用率预期。</p>

<p>***3. ***</p>

<p>性能优化</p>

<p></p>

<p>从上面的写入瓶颈分析，我们发现了 ES 无法将资源用满的原因来自于分布式的长尾问题。于是们着重思考如何消除分布式的长尾问题。然后也在探寻其他的优化点。整体性能优化，我们分成了三方向：</p>

横向优化，优化写入模型，消除分布式长尾效应。

纵向优化，提升单节点写入能力。

应用优化，探究业务节省资源的可能。

<p>这次的性能优化，我们在这三个方向上都取得了一些突破。</p>

<p>****| 3.1 优化写入模型</p>

<hr>

<hr>

<hr>
<hr>
<p>****

<p>**写入模型的优化思路是将一个 BulkRequest 请求，转发到尽量少的 shard，甚至只转发到一个 shard，来减少甚至消除分布式长尾效应。我们完成的写入模型优化，最终能做到一个 BulkRequest 请求只转发到一个 shard，这样就消除了分布式长尾效应。**</p>

<p>写入模型的优化分成两个场景。一个是数据不带 routing 的场景，这种场景用户不依赖数据分布比较容易优化的，可以做到只转发到一个 shard。另一个是数据带了 routing 的场景，用户对数据分布有依赖，针对这种场景，我们也实现了一种优化方案。</p>

<p>3.1.1 不带 routing 场景</p>

<p>由于用户对 routing 分布没有依赖，ClientNode 在处理 BulkRequest 请求中，给 BulkRequest 的一批请求带上了相同的随机 routing 值，而我们生成环境的场景中，一批数据是写入一个索引中，以这一批数据就会写入一个物理 shard 中。</p>

<p></p>

<p>3.1.2 带 routing 场景</p>

<p>**

<p>下面着重介绍下我们在带 routing 场景下的实现方案。这个方案，我们需要在 ES Server 层和 ES SDK 都进行优化，然后将两者综合使用，来达到一个 BulkRequest 上的一批数据写入一个物理 shard 的效果。优化思路 ES SDK 做一次数据分发，在 ES Server 层做一次随机写入来让一批数据写入同一个 shard。</p>

<p>先介绍下 Server 层引入的概念，我们在 ES shard 之上，引入了逻辑 shard 的概念，命名为 <code>number_of_routing_size</code>。ES 索引的真实 shard 我们称之为物理 shard，命名是 <code>number_of_shards</code>。</p>

<p>物理 shard 必须是逻辑 shard 的整数倍，这样一个逻辑 shard 可以映射到多个物理 shard。一逻辑 shard，我们命名为 slot，slot 总数为 <code>number_of_shards / number_of_routing_size /code>。</p>

<p>数据在写入 ClientNode 的时候，ClientNode 会给 BulkRequest 的一批请求生成一个相同的随机值，目的是为了写入的一批数据，都能写入相同的 slot 中。数据流转如图所示：</p>

<p></p>

<p>最终计算一条数据所在 shard 的公式如下：</p>

<p>| slot = hash(random(value)) % (number_of_shards/number_of_routing_size) shard_num = hash(_routing) % number_of_routing_size + number_of_routing_size * slot |</p>

<p>然后我们在 ES SDK 层进一步优化，在 BulkProcessor 写入的时候增加逻辑 shard 参数，在 add 数据的时候，可以按逻辑 shard 进行 hash，生成多个 BulkRequest。这样发送到 Server 的一个 BulkRequest 请求，只有一个逻辑 shard 的数据。最终，写入模型变为如下图所示：</p>

<p></p>

<p>经过 SDK 和 Server 的两层作用，一个 BulkRequest 中的一批请求，写入了相同的物理 shard</p>

<p>这个方案对写入是非常友好的，但是对查询会有些影响。由于 routing 值是对应的是逻辑 shard 一个逻辑 shard 要对应多个物理 shard，所以用户带 routing 的查询时，会去一个逻辑 shard 对应多个物理 shard 中查询。</p>

<p>我们针对优化的是日志写入的场景，日志写入场景的特征是写多读少，而且读写比例差别很大，以在实际生产环境中，查询的影响不是很大。</p>

<p>*****| 3.2 单节点写入能力提升 *****</p>

<hr>

<p>单节点写入性能提升主要有以下优化：</p>

<p>backport 社区优化，包括下面 2 方面：</p>

```
<ul>
<li>merge 社区 flush 优化特性: [#27000] Don't refresh on <code>_flush</code> <code>_for_e_merge</code> and <code>_upgrade</code></li>
<li>merge 社区 translog 优化特性, 包括下面 2 个:
<ul>
<li>[#45765] Do sync before closeIntoReader when rolling generation to improve index performance</li>
<li>[#47790] sync before trimUnreferencedReaders to improve index performance</li>
</ul>
</li>
</ul>
<p>这些特性我们在生产环境验证下来, 性能大概可以带来 18% 的性能提升。</p>
<p>我们还做了 2 个可选性能优化点: </p>
<ul>
<li>优化 translog, 支持动态开启索引不写 translog, 不写 translog 的话, 我们可以不再触发 translog 的锁问题, 也可以缓解了 IO 压力。但是这可能带来数据丢失, 所以目前我们做成动态开关, 需要追数据的时候临时开启。后续我们也在考虑跟 flink 团队结合, 通过 flink checkpoint 保证数据可靠性, 就可以不依赖写入 translog。从生产环境我们验证的情况看, 在写入压力较大的索引上开启不 translog, 能有 10-30% 不等的性能提升。</li>
<li>优化 lucene 写入流程, 支持在索引上配置在 write 线程不同步 flush segment, 解决前面提到尾原因中的 lucene refresh 问题。在生产环境上, 我们验证下来, 能有 7-10% 左右的性能提升。</li>
</ul>
<h3 id="toc_h3_0"></h3>
<p><strong>3.2.1 业务优化</strong></p>
<p>在本次进行写入性能优化探究过程中, 我们还和业务一起发现了一个优化点, 业务的日志数据中在 2 个很大的冗余字段(args、response), 这两个字段在日志原文中存在, 还另外用了 2 个字段存, 这两个字段并没有加索引, 日志数据写入 ES 时可以不从日志中解析出这 2 个字段, 在查询的时候接从日志原文中解析出来。</p>
<p>不清理的冗余字段, 我们验证下来, 能有 20% 左右的性能提升, 该优化同时还带来了 10% 左右存储空间节约。</p>
<p>***4. ***</p>
<p>生产环境性能提升结果</p>
<p></p>
<h3 id="-----4-1-写入模型优化---">*****| <strong><strong>4.1 写入模型优化</strong></strong> **</h3>
<p>**<br>
**</p>
<p>我们重点看下写入模型优化的效果, 下面的优化, 都是在客户端、服务端资源没做任何调整的情况下的生产数据。</p>
<p>下图所示索引开启写入模型优化后, 写入 tps 直接从 50w/s, 提升到 120w/s。</p>
<p></p>
<p>生产环境索引写入性能的提升比例跟索引混部情况、索引所在资源大小(长尾问题影响程度)等因素影响。从实际优化效果看, 很多索引都能将写入速度翻倍, 如下图所示: </p>
<p></p>
<h3 id="-4-2-写入拒绝量-write-rejected-下降-----"><strong><strong><strong><strong><strong>| <strong><strong>4.2 写入拒绝量(write rejected)下降</strong></strong></strong></strong> **** *****</h3>
```

<hr>

<hr>

<p>然后再来看一个关键指标，写入拒绝量(write rejected)。ES datanode queue 满了之后就会出现 rejected。</p>

<p>rejected 异常带来个危害，一个是个别节点出现 rejected，说明写入队列满了，大量请求在队中等待，而 region 内的其他节点却可能很空闲，这就造成了 cpu 整体利用率上不去。</p>

<p>rejected 异常另一个危害是造成失败重试，这加重了写入负担，增加了写入延迟的可能。</p>

<p>优化后，由于一个 bulk 请求不再分到每个 shard 上，而是写入一个 shard。一来减少了写入请求，二来不再需要等待全部 shard 返回。</p>

<p></p>

<p>****|4.3 延迟情况缓解</p>

<p>最后再来看下写入延迟问题。经过优化后，写入能力得到大幅提升后，极大的缓解了当前的延迟情况。下面截取了集群优化前后的延迟情况对比。</p>

<p></p>

<p>***5. ***</p>

<p>总结</p>

<p></p>

<p>这次写入性能优化，滴滴 ES 团队取得了突破性进展。写入性能提升后，我们用更少的 SSD 机器撑了数据写入，支撑了数据冷热分离和大规格存储物理机的落地，在这过程中，我们下线了超过 400 物理机，节省了每年千万左右的服务器成本。在整个优化过程中，我们深入分析 ES 写入各个环节的情况，去探寻每个耗时环节的优化点，对 ES 写入细节有了更加深刻的认识。我们还在持续探寻更的优化方式。而且我们的优化不仅在写入性能上。在查询的性能和稳定性，集群的元数据变更性能等方面也都在不断探索。我们也在持续探究如何给用户提交高可靠、高性能、低成本、更易用的 ES，来会有更多干货分享给大家。</p>

<p>原文：滴滴 Elasticsearch 千级 TPS 写入性能翻倍技术剖析</p>