Elasticsearch Service

ES 内核增强

产品文档

腾讯云

文档目录

ES 内核增强
内核版本发布记录

最近更新时间：2022-06-28 09:36:21

在完全兼容开源内核的基础上，腾讯云 Elasticsearch Service（ES）基于多场景大规模的丰富应用经验，围绕集群性能增强、稳定性提升、成本优化等方向，对 ES 内核进行了持续的深度研究和优化，并始终与开源社区保持着密切的交流。本文为大家介绍腾讯云 ES 的关键内核优化工作。

2022年4月发布的重点优化特性：

优化分类	优化策略	支持版本
性能优化	时序索引查询裁剪优化，时间范围检索从大范围遍历优化为边界定点裁剪，高基维时序范围搜索性能提升10倍+。	7.14.2
	DSL 查询结果支持列式返回，大幅降低重复 key 冗余，网络带宽占用减少35%，性能提升20%。	7.14.2
	节点间数据透传序列化优化，减少冗余的序列化开销，提升查询性能30%。	7.14.2
	X-Pack 鉴权性能优化，通过特殊权限处理、缓存、延迟加载等机制消除 CPU 热点，提升查询性能30%+。	7.10.1、7.14.2
	查询支持细粒度 block 级别采样优化，提升 Top-k、avg、min、max、histogram 等算子预估查询性能10倍+。	7.14.2
功能优化	查询 preference 参数优化，支持 _shards 和 custom_string 组合使用固定主副分片，确保评分场景查询结果稳定。	7.14.2
	Keyword 字段超长内容截断优化，超长内容支持不报异常截断写入。	7.14.2
	查询超时底层细粒度控制优化，避免大查询被主动取消或超时（查询需携带 timeout 参数）后继续占用集群资源。	7.10.1、7.14.2
稳定性优化	修复查询流程中内存限流特定场景内存泄露问题、进一步优化内存限流策略，避免聚合场景打爆内存，提升集群稳定性。	7.14.2
	修复节点脱离集群后长时间反复加入、踢出问题，提升集群稳定性。	7.10.1、7.14.2
	优化节点、索引维度分片均衡策略，提升节点、索引层面的分片均衡能力，消除负载热点。	7.10.1、7.14.2
	优化多盘场景分片搬迁均衡策略，提升多盘场景分片搬迁性能。	6.8.2、7.10.1、

		7.14.2
	优化分片启动、分片失败任务优先级，避免索引长时间不可用。	6.8.2、7.10.1、7.14.2
	集群扩展性优化，大幅提升分片数量、节点扩展能力，元数据变更、集群重启性能倍数级提升。	7.14.2
安全优化	修复 Log4j 安全漏洞。	全版本

2021年2月发布的重点优化特性：

优化维度	优化分类	优化策略	支持版本
性能优化	写入性能优化	分片定向路由优化，解决单索引多分片场景写入流程长尾分片问题，写入吞吐提升10%+，CPU 下降20%+。	6.8.2、7.5.1、7.10.1
	查询性能优化	对查询结果进行裁剪，代替 filter_path，查询性能提升10%+。	6.8.2、7.5.1、7.10.1
稳定性优化	内存优化	大幅降低高并发写入、大查询导致节点卡死、集群雪崩问题，整体可用性提升至 99.99%。 • 基于内存资源的 Netty 网络层高并发写入限流。快速回收查询、写入异常的请求所消耗内存，避免异常请求导致内存堆积。优化自研单个请求熔断器，避免单个大查询占用资源过多。 • 基于 GC 负债管理及时自动重启内存彻底打满的节点。Lucene 文件类型内存映射模型可配置，根据不同的业务场景优化系统内存使用。	6.8.2、7.5.1、7.10.1
	JDK、GC 优化	引入腾讯自研优化的 KONA JDK11，修复已知 JDK bug，提升 FGC 串行回收能力。 切换 G1 垃圾回收器，提升 GC 效率，减少 Old GC 导致的毛刺。	6.8.2、7.5.1、7.10.1
	元数据性能优化	mapping 更新任务优先级优化，解决高并发 mapping 更新任务堵塞导致大量请求堆积打挂节点问题。元数据异步落盘优化，提升元数据同步性能，避免索引创建、mapping 更新频繁超时。	6.8.2、7.5.1、7.10.1
成本优化	存储优化	引入 zstd 压缩算法，压缩比提升30% - 50%，压缩性能提升30%。	6.8.2、7.5.1、7.10.1

截至2020年7月腾讯云 ES 团队自启动内核研究至今的重点优化特性：

优化维度	优化分类	优化策略	支持版本

性能优化	写入性能优化	Translog 锁机制优化，总体写入性能提升20%。写入去重优化，segment 文件裁剪优化，带主键写入性能提升50%+。	7.5.1、7.10.1
	查询性能优化	• 聚合性能优化，查询高效剪枝，排序场景 composite 聚合性能提升3 - 7倍。 • 查询缓存优化，取消开销高、命中率低的数据缓存，实际场景查询毛刺从750ms降至50ms。 • 合并策略优化，自研基于时序、大小相似性分层合并策略、冷分片自动合并策略，搜索场景查询性能提升40%+。 • 查询 Fetch 阶段顺序抓取优化，提升缓存命中率，查询结果集较大场景，性能提升10%+。	6.4.3、6.8.2、7.5.1、7.10.1
稳定性优化	可用性优化	• 接入层曲线平滑限流。 • 协调节点汇聚子查询结果反序列化膨胀预估、内存检查。 • 大聚合查询结果集流式检查，内存达到阈值熔断请求。 • 自研单个请求熔断器（Single Request Circuit Breaker），避免单个大查询占用资源过多，避免单个大查询占用资源过多影响其它查询。 • 大幅降低高并发写入、大查询导致节点卡死、集群雪崩问题，整体可用性提升至99.99%。	6.4.3、6.8.2、7.5.1、7.10.1
	均衡策略优化	• 引入基于索引、节点打散的均衡策略，优化集群新增节点导致分片严重不均问题。 • 优化多盘（多数据目录）之间分片不均问题。 • 提升集群扩容场景、多盘场景新建索引分片均衡性，减少人工运维成本。	5.6.4、6.4.3、6.8.2、7.5.1、7.10.1
	滚动重启速度优化	• 优化节点重启分片复用本地数据逻辑。 • 精准控制预定延时时间内的分片拷贝恢复。 大集群单节点重启时间从10多分钟降至1分钟。	6.4.3、6.8.2、7.5.1、7.10.1
	在线切主	自研在线切主功能，用户通过 API 指定偏好 master，实现秒级在线切换，典型使用场景： • 人工运维时发现当前 master 高负载，在线切换 master 至规格更高、负载低的节点。 • 滚动重启时，master 节点放到最后重启，重启之前先将 master 角色快速切到别的节点再重启，服务影响从分钟级缩短到秒级。	6.4.3、6.8.2、7.5.1、7.10.1
成本优化	内存优化	• 自研堆外 cache，实现 FST Off-Heap 优化。 • 堆外 cache 保障 FST 回收策略可控。 • 精准淘汰策略提高 cache 命中率。 • 零拷贝加多级 cache 保障访问性能。 • 大幅降低堆内存开销，GC 时长下降10%+，单节点磁盘管理规模可达50TB，读写性能基本不受影响。	6.8.2、7.5.1、7.10.1

	存储优化	• 自研 ID 字段行存裁剪，时序场景存储开销降低20%+。	5.6.4、6.4.3、6.8.2、7.5.1、7.10.1

定向路由优化

最近更新时间：2021-05-18 16:02:48

背景

在规模较大的集群中（100+节点），单个索引一般有超多个分片配置（100+）。

用户一般采用 bulk 写入，ES 默认采用 _id 作为单个文档写入的 routing，路由打散分片。这样一个 bulk 请求将会被均匀拆分打散为分片数量的子写入请求，发送给每个分片执行写入，协调节点需要等待所有分片写入完毕才会返回给客户端。当分片数过多时，就容易出现长尾子请求，即有可能部分子请求因节点故障或 Old GC、网络抖动等延迟响应，导致整个 bulk 请求响应缓慢而堆积，最终导致节点写入队列打满出现写入拒绝。另一方面，拆分过多的子请求无法提升数据节点写入吞吐，无法充分利用 CPU。

优化方案

在多分片 bulk 写入场景，通过 routing 的方式实现一个 bulk 只写入到一个分片，降低网络开销、提升数据节点 CPU 使用率、避免长尾分片影响整个 bulk 请求。

ES 内核提供索引属性，可以为一个 bulk 请求的写入子请求统一自动添加一个随机 routing，确保子请求只路由到一个分片、且确保索引各个分片数据均衡。

使用方法

新增的索引属性名称：**index.bulk_routing.enabled**，默认值为 false，索引创建时可以指定，也可以后续动态更新。

新建索引时指定开启 bulk routing 优化：

```
curl -X PUT "localhost:9200/my-index" -H 'Content-Type: application/json' -d'
{
"settings" : {
"index" : {
"bulk_routing.enabled" : true
}
}
}
'
```

动态更新单个索引方法：

```
curl -X PUT "localhost:9200/my-index/_settings?pretty" -H 'Content-Type: applicat
ion/json' -d'
{
"index.bulk_routing.enabled": true
}
'
```

一般会为同一类业务的多个索引创建模板，在索引滚动场景可批量生效：

```
curl -X PUT "localhost:9200/_template/bulk_routing_template?pretty" -H 'Content-T
ype: application/json' -d'
{
"index_patterns": ["indices-prefix*"],
"settings": {
"bulk_routing.enabled": true
}
}
'
```

优化限制

写入限制

开启了 bulk routing 优化，只有在如下情况才会将同一索引的子请求路由到一个分片：

- 用户写入时没有自定义 routing。
- 用户没有写入的单条文档没有自定义 _id 字段。

如果有以上情况，该 bulk 请求无法优化，因为会优化会和用户自定义的 routing 和 _id 冲突。

查询限制

开启优化后，会自动为 bulk 请求中的子请求添加随机 routing。普通的查询无任何影响，但通过 id 获取单条文档（getById）将会受到影响，因为 ES 目前 getById 的实现是默认通过 _id routing 分片。不过这种问题在用户写入时自定义 routing 场景也同样存在。此时 getById 只能同时携带正确的 routing 才能获取单条原始的文档信息，routing 可以通过普通 query 查询获取。

场景限制

该优化项主要针对多节点、单个索引多分片效果明显，在节点数较少、分片数较少场景例如小于10个分片，该优化可能效果不明显。

优化效果

目前在线上大客户集群验证（100+节点、单索引100+分片），开启多分片场景 bulk routing 优化，拒绝直接降为0，CPU 下降25%，写入速度上升10%。

支持版本

6.8.2、7.5.1、7.10.1

压缩算法优化

最近更新时间：2021-09-24 17:22:57

背景

Lucene 当前针对 Document Fields 数据的存储，支持两种压缩算法：

- LZ4
- Deflate

LZ4 具有更快的压缩与解压速度，而 Deflate 在压缩率上更占优势。两者在性能与压缩率上存在明显的差异，基于现有的压缩算法，用户不能很好的兼容压缩比和性能，Lucene 默认的压缩算法是 LZ4。

优化方案

整合业内先进的压缩算法 Zstandard（ZSTD），提升压缩率的同时，性能损耗小。

Zstandard 压缩算法的优势

Zstandard 压缩算法可以说**兼顾了 LZ4 与 Deflate 两者的优点：**在性能上与 LZ4 相当（针对日志数据的测试中，发现 Zstandard 算法比 LZ4 略优），而压缩率略弱于 Deflate。

如下是关于三种压缩算法的对比测试结果：

压缩算法	加载时间(1 Shard)	加载时间(5 Shards)	Fields(*fdt) 文件大小	索引总大小
LZ4	1143769ms	420447ms	4.15 GB	6.3 GB
Deflate	1270408ms	448738ms	2.56 GB	4.7 GB
Zstandard(16K Chunk)	1109414ms	415256ms	2.93 GB	5.1 GB
Zstandard(32K Chunk)	1088959ms	406661ms	2.67 GB	4.8 GB

注意：

1. 测试数据：基于某典型日志应用类数据。
2. 测试方法：基于 Elasticsearch Rest High Level Client API。

使用方式

基于 Rest High Level Client API

在创建 Index 时，为 CreateIndexRequest 添加"index.codec"配置项，value 设置为"zstandard"：

```
CreateIndexRequest createRequest = new CreateIndexRequest(indexName);
createRequest.settings(Settings.builder()
.put("index.number_of_shards", shards)
.put("index.number_of_replicas", replicas)
.put("index.codec", "zstandard")
);
```

基于 HTTP 请求

类似的，也在 settings 中添加"index.codec"配置项，并将 value 设置为"zstandard"：

```
PUT /newIndex
{
"settings": {
"index.codec": "zstandard",
"index.number_of_shards": 1
}
}
```

优化效果

ZSTD 行存压缩率相较 LZ4 提升35%，性能和 LZ4 相当。

支持版本

6.8.2、7.5.1、7.10.1

FST Off Heap 内存优化

最近更新时间：2021-05-18 16:02:49

背景

ES 单节点上，倒排索引中的 FST 结构默认常驻堆内内存，占比较高，尤其是在大磁盘的冷节点上，占比可达 50%+，制约了单节点管理磁盘的能力，堆内内存受限，影响节点可用性。而在冷节点上，查询请求非常少，FST 常驻内存意义不大，因此我们需要将该部分数据结构移动到堆外管理，默认不加载，需要时从磁盘加载到堆外直接使用，以降低堆内内存使用量，提升单节点磁盘管理能力。

优化方案

基于 WLFU 淘汰策略，实现精准控制的堆外 cache，堆内基于零拷贝、弱引用实现第二级 cache，性能和堆内访问基本持平。

使用方式

开启、关闭 Off Heap 功能（默认关闭）

```
curl -H "Content-Type:application/json" -XPUT http://localhost:9200/_cluster/sett
ings -d '{
"persistent" : {
"indices.segment_memory.off_heap.enable" : true
}
}'
```

调整 Off Heap Cache 大小（默认500MB）

```
curl -H "Content-Type:application/json" -XPUT http://localhost:9210/_cluster/sett
ings -d '{
"persistent" : {
"indices.segment_memory.off_heap.size" : "5gb"
}
}'
```

可以设置为单节点堆外内存的1/3，最大不要超过32GB。具体示例如下：

- 单节点总内存（包括 jvm 和堆外内存）共64GB，可以设置为（64-32）/3 = 10GB

- 单节点总内存（包括 jvm 和堆外内存）共96GB，可以设置为（96-32）/3 = 20GB

优化效果

内存开销、数据管理能力、GC 优势明显，性能持平略有优势。

方案对比	FST **存放位置**	FST 内存占用量	**单个 FST 堆内存占用量**	**单节点最大磁盘数据量**
原生方案	堆内内存	全量存储在内存中，内存占用量大	MB 级别（源生 FST 数据结构）	10TB（需调优）
优化方案	堆外内存	Cache LRU 淘汰冷数据 内存占用量小	100Byte左右（Cache Key 大小）	50TB

写入性能对比	内存使用量（MB）	GC 时长（s）	TPS	90%时延（ms）	99%时延（ms）
原生方案	402.59	20.453	198051	463.201	617.701
优化方案	102.217	18.969	201188	455.124	618.379
Diff	优74.6%	优7.26%	优1.58%	优1.74%	劣0.11%

查询性能对比	**内存使用量（MB）**	**GC 时长（s）**	**QPS**	90%时延（ms）	99%时延（ms）
原生方案	401.806	20.107	200.057	3.96062	11.1894
优化方案	101.004	19.228	200.087	3.87805	11.2316
Diff	优74.9%	优4.37%	-	优2.00%	劣0.38%

支持版本

6.8.2、7.5.1、7.10.1