Elasticsearch Service

ES Kernel Enhancement

Product Documentation

Contents

ES Kernel Enhancement
Kernel Release Notes

Last updated：2022-06-28 09:36:21

Tencent Cloud Elasticsearch Service (ES) team has been continuously optimizing the ES kernel based on its extensive practical experience in large-scale applications while remaining fully compatible with the open-source Elasticsearch kernel, in an effort to improve cluster performance and stability and reduce costs. In addition, the team keeps up with latest updates in the community. This document describes the major kernel optimizations of ES.

Major optimizations in **April 2022**:

Optimization Category	Optimization Policy	Supported Version
Performance	Time series index query clipping is optimized, shifting from large-scale traversal to fixed-point boundary clipping and increasing the high-dimensional time series search performance by over ten times.	7.14.2
	DSL query results can be returned in columns, which greatly reduces the duplicate key redundancy, lowers the network bandwidth usage by 35%, and increases the performance by 20%.	7.14.2
	The serialization of transparent data transfer between nodes is optimized, reducing the redundant serialization costs and increasing the query performance by 30%.	7.14.2
	The X-Pack authentication performance is optimized. CPU hotspots are eliminated through special permission processing, caching, and delayed loading, improving the query performance by over 30%.	7.10.1, 7.14.2
	The query performance is optimized in fine-grained block-level sampling, increasing the estimated query performance of operators such as topk, avg, min, max, and histogram by over ten times.	7.14.2
Feature	The query preference parameters are optimized. `_shards` and `custom_string` can be used in combination to fix primary and replica shards, which ensures stable query results in scoring scenarios.	7.14.2
	The truncation of super-long content of keyword fields is optimized, so that such content can be written without an truncation exception reported.	7.14.2
	The underlying fine-grained control of query timeout is optimized to avoid the further occupation of cluster resources by a large number of canceled or timed-out queries (the queries should carry the `timeout` parameter).	7.10.1, 7.14.2

Stability	The memory leak issue in specific memory usage throttling scenarios during the query process is fixed, and the memory usage throttling policy is further optimized to avoid OOM errors in aggregation scenarios and enhance the cluster stability.	7.14.2
	The issue of repeated join and removal of nodes leaving the cluster is fixed to increase the cluster stability.	7.10.1, 7.14.2
	The node-level and index-level shard balancing policies are optimized to improve the shard balancing capabilities and eliminate load hotspotting.	7.10.1, 7.14.2
	The shard relocation and balancing policies are optimized for multi-disk scenarios to improve the shard relocation performance.	6.8.2, 7.10.1, 7.14.2
	The shard start and the priority of failed shard tasks are optimized to avoid prolonged index unavailability.	6.8.2, 7.10.1, 7.14.2
	The cluster scalability performance is optimized, with the shard quantity and node expansion capabilities greatly increased, many metadata changes implemented, and cluster restart performance multiplied.	7.14.2
Security	The Log4j vulnerability is fixed.	All versions

Major optimizations in **February 2021**:

Optimization Dimension	Optimization Category	Optimization Policy	Supported Version
Performance	Write performance	Shard-targeted routing is optimized, solving the long-tail shard issue in the writing process in single-index multi-shard scenarios. This also increases the write throughput by over 10% and reduces the CPU usage by over 20%.	6.8.2, 7.5.1, 7.10.1
	Query performance	Query performance is improved by over 10% by cropping the query results, instead of using `filter_path`.	6.8.2, 7.5.1, 7.10.1
Stability	Memory	Node crashes and cluster avalanches caused by high-concurrent writes and large queries are significantly reduced, and the overall availability is increased to 99.99%. • High-concurrent write traffic is limited at the Netty network layer based on memory resources. The memory consumed by query and write request exceptions is quickly repossessed to	6.8.2, 7.5.1, 7.10.1

		avoid memory leak. The proprietary single request circuit breaker is optimized to prevent a large query from occupying excessive resources. • Based on GC management, nodes with completely occupied memory are automatically restarted in time. The Lucene file type memory mapping model can be configured to improve the system's memory usage in different business scenarios.	
	JDK, GC	Tencent's proprietary KONA JDK11 is adopted and known JDK bugs are fixed, improving serial full GC capability. You can switch to the G1 collector to improve GC efficiency and reduce glitches caused by old GC.	6.8.2, 7.5.1, 7.10.1
	Metadata performance	The priority of mapping update tasks is optimized, solving the issue where nodes cannot work properly due to excessive requests in the queue caused by high number of concurrent mapping update tasks. Metadata asynchronous storage is optimized and metadata synchronization performance is improved to avoid frequent timeouts of index creations and mapping updates.	6.8.2, 7.5.1, 7.10.1
Costs	Storage	The zstd compression algorithm is adopted, increasing the compression ratio by 30% to 50% and the compression performance by 30%.	6.8.2, 7.5.1, 7.10.1

Major optimizations as of **July 2020** since the ES team restarted its kernel research:

Optimization Dimension	Optimization Category	Optimization Policy	Supported Versions
Performance	Write performance	The translog lock mechanism is optimized, increasing the overall write performance by 20%. Write deduplication and segment file cropping are optimized, increasing the performance of writes with primary keys by over 50%.	7.5.1, 7.10.1
	Query performance	• The aggregation performance is optimized, making query pruning more efficient and improving the composite aggregation performance by 3 to 7 times in sorting scenarios. • The query cache is optimized by canceling data caches with high overheads and low hit rates, reducing query glitches from 750 ms to 50 ms in actual use cases. • The merge policies are optimized by developing proprietary merge policies based on time series and size similarity and auto warm shard merge policy, improving the query performance by over 40% in search scenarios.	6.4.3, 6.8.2, 7.5.1, 7.10.1

		• Sequence capture in the query fetch phase is optimized, increasing the cache hit rate and improving the performance by over 10% in scenarios where the result set is large.	
Stability	Availability	• Traffic can be limited through a smooth line curve at the access layer. • The coordinator node performs memory bloat estimation after receiving results returned by the data node to check whether the estimated memory will exceed the limit. • Result sets of large aggregated queries are checked in a streaming manner, and requests will be canceled if the used memory reaches the threshold. • The proprietary single request circuit breaker can prevent a large query from occupying excessive resources and thus affecting other queries. • Node crashes and cluster avalanches caused by high-concurrence writes and large queries are significantly reduced, and the overall availability is increased to 99.99%.	6.4.3, 6.8.2, 7.5.1, 7.10.1
	Balancing policy	• Balancing policies based on index and node distribution are introduced, alleviating the serious uneven allocation of shards caused by new nodes added to the cluster. • The uneven allocation of shards among multiple disks (multiple data directories) is alleviated. • The balance of shards of newly created indices in cluster scale-out scenarios and multiple-disk scenarios is improved, reducing Ops costs.	5.6.4, 6.4.3, 6.8.2, 7.5.1, 7.10.1
	Rolling restart speed	• The logic of reusing local data for shards in case of node restart is optimized. • The restoration of shard copies within a scheduled delay time period can be precisely controlled. The time to restart one single node in a large cluster is reduced from over 10 minutes to 1 minute.	6.4.3, 6.8.2, 7.5.1, 7.10.1
	Online master switch	The proprietary online master switch feature allows you to switch the master online in seconds by specifying the preferred master through APIs. Typical use cases include: • You can switch online from the current heavily loaded master to a node with a higher specification and a lower load during manual Ops. • During rolling restart, you can restart the master node last and quickly switch the master role to another node before the restart, which helps reduce the service interruption from minutes to seconds.	6.4.3, 6.8.2, 7.5.1, 7.10.1

Costs	Memory	• The proprietary off-heap cache helps achieve FST off-heap optimization. • The off-heap cache ensures that the FST reclaim policy is controllable. • The precise eviction policy improves the cache hit rate. • Zero-copy and multi-level caches guarantee high access performance. • The heap memory overheads are significantly reduced, the GC time is decreased by over 10%, and the disk capacity of a single node can reach 50 TB, with read/write performance generally unaffected.	6.8.2, 7.5.1, 7.10.1
	Storage	• The proprietary ID field-based row storage cropping algorithm reduces storage overheads by over 20% in time series scenarios.	5.6.4, 6.4.3, 6.8.2, 7.5.1, 7.10.1

Targeted Routing Optimization

Last updated：2021-07-01 10:02:56

Background

In a larger cluster (with 100+ nodes), a single index generally has many shards (100+).

Users generally write in bulk, and ES uses `\_id` as the routing for writing a single document by default, so that shards can be distributed through routing. Such a bulk request will be evenly split into write subrequests of the number of shards, which will be then sent to each shard for writing. The coordinator node needs to wait for all shards to be written before returning to the client. If the number of shards is too large, long-tail subrequests appear easily, that is, some subrequests may be delayed in responding due to node failures, Old GC, network jitters, etc., resulting in the slow response and heap of the entire bulk request and eventually causing the node write queue to fill up. At this time, write rejection will occur. Moreover, splitting one bulk request into too many subrequests cannot increase the write throughput of data nodes and cannot make full use of the CPU.

Optimized Scheme

In the multi-shard bulk write scenario, one bulk request is written to only one shard through routing, which reduces the network overheads, increases the CPU utilization of data nodes, and prevents long-tail shards from affecting the entire bulk request.

The ES kernel provides an index attribute that can uniformly and automatically add a random routing for each write subrequest of a bulk request, ensuring that one subrequest is only routed to one shard and that the data of each shard is balanced in the index.

Directions

The new index attribute is **index.bulk_routing.enabled**, and its default value is `false` , which can be specified during index creation or dynamically updated subsequently.

Specify to enable bulk routing optimization when creating an index:

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

```
"bulk_routing.enabled" : true
}
}
}
'
```

Dynamically update a single index:

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

Generally, a template can be created for multiple indexes of the same business type, which can take effect in batches in index rolling scenarios:

```
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
}
}
'
```

Optimization Limits

Write limits

After bulk routing optimization is enabled, subrequests of the same index will be routed to the same shard only in the following situations:

- Users don't customize the routing during writes.
- Users don't customize the `\_id` field for a single document.

In the above situations, the bulk request cannot be optimized because the optimization will conflict with the user-defined routing and `\_id` .

Query limits

After the optimization is enabled, a random routing is automatically added for each subrequests of a bulk request. This doesn't affect general queries at all. However, it affects queries for getting a single document by ID (getById), because ES' current implementation of `getById` uses `\_id` to route shards by default. This problem also exists in the scenarios where users customize the routing during writes. In this case, `getById` can only get the original document information by carrying the correct routing at the same time, which can be obtained through ordinary queries.

Scenario limits

This optimization item works better in scenarios where there are many nodes and each index has many shards. Its optimization effect may not be obvious in scenarios with a small number of nodes and a small number of shards per index, for example, less than 10 shards.

Optimization Effect

Testing in online large customer clusters (with 100+ nodes and 100+ shards per index) shows that after the bulk routing optimization is enabled, the rejections are directly reduced to 0, the CPU utilization is decreased by 25%, and the write speed is increased by 10%.

Supported Versions

6.8.2, 7.5.1, and 7.10.1

Compression Algorithm Optimization

Last updated：2021-09-24 17:21:29

Background

Lucene currently supports two compression algorithms for storing document fields data:

- LZ4
- Deflate

LZ4 has a higher compression and decompression speed, while Deflate has a higher compression ratio. They have obvious differences in performance and compression ratio. Based on these two existing compression algorithm, you cannot get a good balance between compression ratio and performance. Lucene uses LZ4 by default.

Optimized Scheme

ES integrates the industry-leading advanced compression algorithm Zstandard (ZSTD) to improve the compression ratio while reducing the performance loss.

Strengths of Zstandard compression algorithm

The Zstandard compression algorithm **has the strengths of both LZ4 and Deflate**: its performance is equivalent to that of LZ4 (tests with log data show that Zstandard is slightly better than LZ4), while its compression ratio is only slightly lower than that of Deflate.

The following are the comparison results of the three compression algorithms:

Compression Algorithm	Load Time (1 Shard)	Load Time (5 Shards)	Fields(*fdt) File Size	Total Index Size
LZ4	1143769 ms	420447 ms	4.15 GB	6.3 GB
Deflate	1270408 ms	448738 ms	2.56 GB	4.7 GB
Zstandard(16K Chunk)	1109414 ms	415256 ms	2.93 GB	5.1 GB
Zstandard(32K Chunk)	1088959 ms	406661 ms	2.67 GB	4.8 GB

Note：

1. Test data: based on a typical log application.
2. Test method: based on Elasticsearch REST High Level Client API.

Directions

Based on REST High Level Client API

When creating an index, add the `index.codec` configuration item for `CreateIndexRequest` and set the value to `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")
);
```

Based on HTTP request

Similarly, add the `index.codec` configuration item in `settings` and set the value to `zstandard`：

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

Optimization Effect

ZSTD has a 35% higher row storage compression ratio than LZ4 and a performance comparable to that of LZ4.

Supported Versions

6.8.2, 7.5.1, and 7.10.1

FST Off-Heap Memory Optimization

Last updated：2021-07-01 10:02:55

Background

On a single ES node, the FST structure in the inverted index resides permanently in the heap memory, which uses a relatively high proportion of the memory (up to over 50% especially on cold nodes with large disks). This restricts the capability of a single node to manage disks, limits the heap memory, and thus affects the node availability. As there are very few query requests on cold nodes, storing FST in the memory is of little significance. Therefore, we need to move this part of data structure out of the heap for management, so that it is not loaded by default but loaded from the disk off heap for direct use when needed, reducing the heap memory usage and improving the single-node disk management capabilities.

Optimized Scheme

ES implements a precisely controlled off-heap cache based on the WLFU elimination strategy and implements a second-level in-heap cache based on zero copy and weak reference. In this way, the performance is comparable to that of in-heap access.

Directions

Enabling/Disabling off-heap feature (disabled by default)

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

Adjusting the size of off-heap cache (500 MB by default)

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

```
}
}'
```

It can be set to 1/3 of the off-heap memory of a single node and should not exceed 32 GB. Below are specific examples:

- If the total memory of s single node (including JVM and off-heap memory) is 64 GB, it can be set to (64 – 32)/3. You can set it to 10 GB.
- If the total memory of s single node (including JVM and off-heap memory) is 96 GB, it can be set to (96 – 32)/3. You can set it to 20 GB.

Optimization Effect

The optimized scheme works much better in terms of memory overheads, data management, and GC and has a slightly better performance.

Scheme	FST **Storage Location**	FST Memory Usage	**Single-FST Heap Memory Usage**	**Single-Node Maximum Disk Data Volume**
Native scheme	Heap memory	Full storage in memory with high memory usage	MB-level (native FST data structure)	10 TB (tuning required)
Optimized scheme	Off-heap memory	Cold data elimination based on cache LRU with low memory usage	Around 100 bytes (cache key size)	50 TB

Write Performance	Memory Usage (MB)	GC Time (s)	TPS	90th Percentile Latency (ms)	99th Percentile latency (ms)
Native scheme	402.59	20.453	198051	463.201	617.701
Optimized scheme	102.217	18.969	201188	455.124	618.379
Difference	74.6% better	7.26% better	1.58% better	1.74% better	0.11% worse

Query Performance	**Memory Usage (MB)**	**GC Time (s)**	**QPS**	90th Percentile Latency (ms)	99th Percentile Latency (ms)

Query Performance	**Memory Usage (MB)**	**GC Time (s)**	**QPS**	90th Percentile Latency (ms)	99th Percentile Latency (ms)
Native scheme	401.806	20.107	200.057	3.96062	11.1894
Optimized scheme	101.004	19.228	200.087	3.87805	11.2316
Difference	74.9% better	4.37% better	-	2.00% better	0.38% worse

Supported Versions

6.8.2, 7.5.1, and 7.10.1