

ElasticSearch

10. 색인 성능 최적화

11. 검색 성능 최적화

12. ElasticSearch 클러스터 구축 시나리오

이상민

00 목차

01 색인 성능 최적화

01 정적 매핑 적용하기

02 refresh_interval 변경하기

03 bulk API

04 그 외의 색인 성능을 확보하는 방법들

01 색인 성능 최적화

01 정적 매팅 적용하기

01 색인 성능 최적화

이전 장에서는 ElasticSearch를 클러스터 환경으로 구축하고 노드를 추가하여 색인 성능을 높였다.

이번에는 ElasticSearch의 설정을 변경함으로써 색인 성능을 향상시키는 방법에 대해 살펴보자.

01 색인 성능 최적화

1. 정적 매팅 적용하기

이번 절에서는 매팅 정보를 생성하는 방법에 따라 발생하는 성능 차이에 대해서 알아보자.

01 색인 성능 최적화

1. 정적 맵핑 적용하기

맵핑 정보를 생성하는 방법

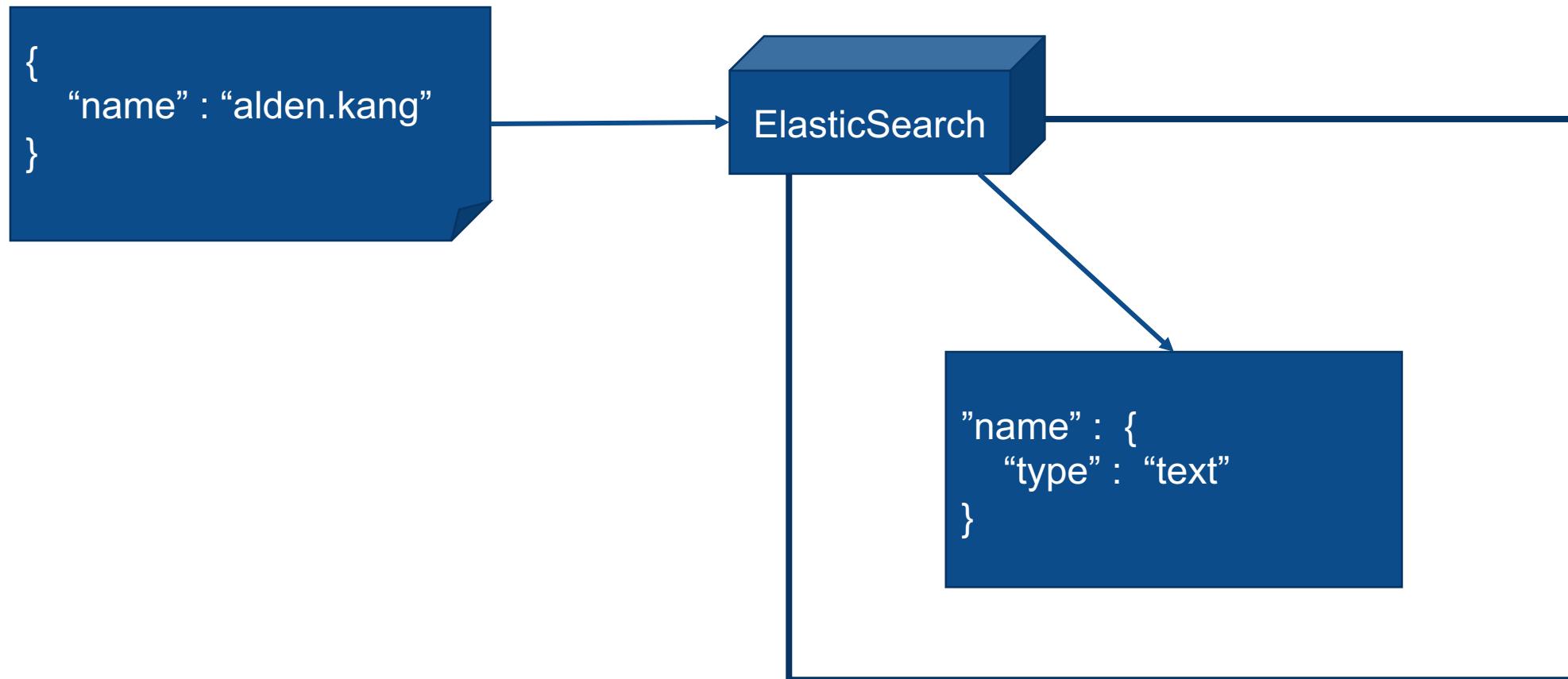
동적 맵핑(dynamic mapping)

정적 맵핑(static mapping)

01 색인 성능 최적화

1. 정적 매팅 적용하기

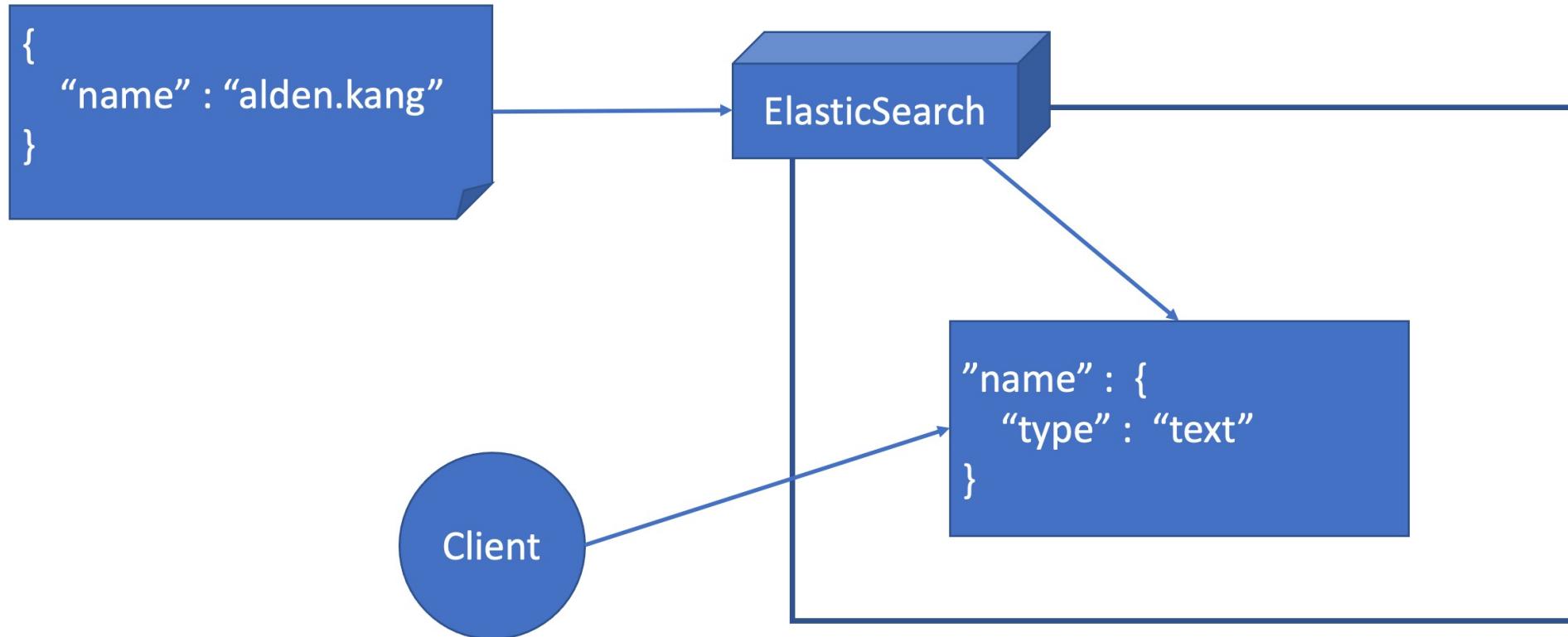
동적 매팅은 매팅 정보를 동적으로 생성하는 것이다. ⇒ 최초 색인되는 문서를 기준으로 매팅 정보를 만들어 낸다.



01 색인 성능 최적화

1. 정적 맵핑 적용하기

정적 맵핑은 맵핑 정보를 미리 만들어 두는 것이다.



01 색인 성능 최적화

1. 정적 매팅 적용하기

두 가지 매팅 정보 생성 방법의 장단점을 정리하면 다음과 같다.

방식	장점	단점
동적 매팅 (Dynamic mapping)	미리 매팅 정보를 생성하지 않아도 된다.	불필요한 필드가 생성될 수 있다.
정적 매팅 (Static mapping)	필요한 필드만 정의해서 사용할 수 있다.	미리 매팅 정보를 생성해야 한다.

- 불필요한 매팅 정보는 불필요한 색인 작업을 유발하게 되어 색인 성능을 저하시킬 수 있다.

01 색인 성능 최적화

1. 정적 매팅 적용하기

문자열 필드는 **text** 와 **keyword** 타입으로 나눌 수 있는데,

간단한 테스트를 통해서 자세히 살펴보자.

두 타입은 분석 방법이 다르기 때문에 성능에도 차이가 있다.

01 색인 성능 최적화

1. 정적 맵핑 적용하기

동적 맵핑 예제

```
POST http://localhost:9200/string_index/_doc?pretty
{
  "mytype" : "String Data Type"
}
```

문서 색인 요청

```
GET http://localhost:9200/string_index/_mapping?pretty
```

색인 맵핑 정보 요청

```
{
  "string_index": {
    "mappings": {
      "properties": {
        "mytype": {
          "type": "text",
          "fields": {
            "keyword": {
              "type": "keyword",
              "ignore_above": 256
            }
          }
        }
      }
    }
  }
}
```

string_index의 맵핑 정보

01 색인 성능 최적화

1. 정적 맵핑 적용하기

정적 맵핑 예제

```
PUT http://localhost:9200/keyword_index?pretty
{
  "mappings": {
    "properties": {
      "mytype": { // 필드 이름
        "type": "keyword" // 데이터 타입
      }
    }
  }
}
```

정적 맵핑 요청

GET http://localhost:9200/keyword_index/_mappings?pretty

색인 맵핑 정보 요청

```
{
  "keyword_index": {
    "mappings": {
      "properties": {
        "mytype": {
          "type": "keyword"
        }
      }
    }
  }
}
```

keyword_index의 맵핑 정보

01 색인 성능 최적화

실습

1. 정적 매팅 적용하기

동적 매팅과 정적 매팅의 성능 비교

100,000 건의 예제 데이터를 다운로드 받는다.

```
curl https://raw.githubusercontent.com/benjamin-btn/ES-SampleData/master/sample10-1.json -o sample.json
```

01 색인 성능 최적화

실습

1. 정적 맵핑 적용하기

동적 맵핑과 정적 맵핑의 성능 비교

```
{  
    "title": "ElasticSearch Training Book",  
    "publisher": "insight",  
    "ISBN": "9788966264849",  
    "release_date": "2020/09/30",  
    "description": "ElasticSearch is cool ..."  
},  
...
```

예제 데이터

01 색인 성능 최적화

실습

1. 정적 매팅 적용하기

동적 매팅과 정적 매팅의 성능 비교

```
time curl -X POST 'localhost:9200/dynamic/_doc/_bulk?pretty' \
-s -H 'Content-Type:application/x-ndjson' --data-binary @sample.json > /dev/null
```

문서 저장 명령어 (bulk API를 사용하는 예제이며, bulk는 뒤 쪽에서 자세히 살펴보자)

01 색인 성능 최적화

실습

1. 정적 매팅 적용하기

동적 매팅과 정적 매팅의 성능 비교

```
0.05s user      // 사용자 모드에서 소요된 CPU 시간  
0.11s system    // 커널 모드에서 소요된 CPU 시간  
1% cpu          // 할당된 CPU 비율  
15.084 total    // 총 처리 시간
```

동적 매팅의 색인 소요 시간

100,000개의 데이터를 동적 매팅을 통해 측정한 총 색인 시간은 **15초** 이다.

01 색인 성능 최적화

1. 정적 맵핑 적용하기

동적 맵핑과 정적 맵핑의 성능 비교

```
GET http://localhost:9200/dynamic/_mappings?pretty
```

동적 맵핑을 통해 생성된 맵핑 정보 조회 요청

```
"ISBN": {  
    "type": "text",  
    "fields": {  
        "keyword": {  
            "type": "keyword",  
            "ignore_above": 256  
        }  
    }  
},  
"description": {  
    "type": "text",  
    "fields": {  
        "keyword": {  
            "type": "keyword",  
            "ignore_above": 256  
        }  
    }  
},  
"publisher": {  
    "type": "text",  
    "fields": {  
        "keyword": {  
            "type": "keyword",  
            "ignore_above": 256  
        }  
    }  
},  
"release_date": {  
    "type": "date",  
    "format": "..."  
},  
"title": {  
    "type": "text",  
    "fields": {  
        "keyword": {  
            "type": "keyword",  
            "ignore_above": 256  
        }  
    }  
}
```

실습

01 색인 성능 최적화

실습

1. 정적 맵핑 적용하기

동적 맵핑과 정적 맵핑의 성능 비교

이번에는 정적 맵핑 방식으로 진행해보자.

```
PUT http://localhost:9200/static
```

```
{
  "mappings": {
    "properties": {
      "title": {
        "type": "keyword"
      },
      "publisher": {
        "type": "keyword"
      },
      "ISBN": {
        "type": "keyword"
      },
      "release_date": {
        "type": "date",
        "format": "yyyy/MM/dd HH:mm:ss | yyyy/MM/dd | epoch_millis"
      },
      "description": {
        "type": "text"
      }
    }
  }
}
```

01 색인 성능 최적화

실습

1. 정적 매팅 적용하기

동적 매팅과 정적 매팅의 성능 비교

```
time curl -X POST 'localhost:9200/_static/_doc/_bulk?pretty' \
-s -H 'Content-Type:application/x-ndjson' --data-binary @sample.json > /dev/null
```

문서 저장 명령어

01 색인 성능 최적화

실습

1. 정적 매팅 적용하기

동적 매팅과 정적 매팅의 성능 비교

**0.04s user
0.14s system
1% cpu
10.909 total**

정적 매팅의 색인 소요 시간

01 색인 성능 최적화

1. 정적 매팅 적용하기

동적 매팅과 정적 매팅의 성능 비교

- 동적 매팅으로 매팅 정보를 생성하면 **text** 필드와 **keyword** 필드로 다중 필드로 적용이 된다.
 - **text** 타입은 입력된 문자열을 쪼개어 역 색인(inverted index) 구조로 만들어서 저장한다.
 - **keyword** 타입은 입력된 문자열을 하나의 토큰으로 저장한다.
- 즉, 문자열 필드가 많으면 많을수록 분석이 필요없는 필드는 **keyword** 타입으로 변경해서 성능을 향상시킬 수 있다.

01 색인 성능 최적화

02 refresh_interval 변경하기

01 색인 성능 최적화

2. refresh_interval 변경하기

refresh_interval 에 대해서 살펴보기 전에 **refresh** 를 먼저 살펴보자.

01 색인 성능 최적화

2. refresh_interval 변경하기

ElasticSearch는 색인되는 문서들을 메모리 버퍼 캐시에 먼저 저장한 후 특정 조건이 되면

메모리 버퍼 캐시에 저장된 색인 정보나 문서들을 디스크에 세그먼트 단위로 저장한다.

01 색인 성능 최적화

2. refresh_interval 변경하기

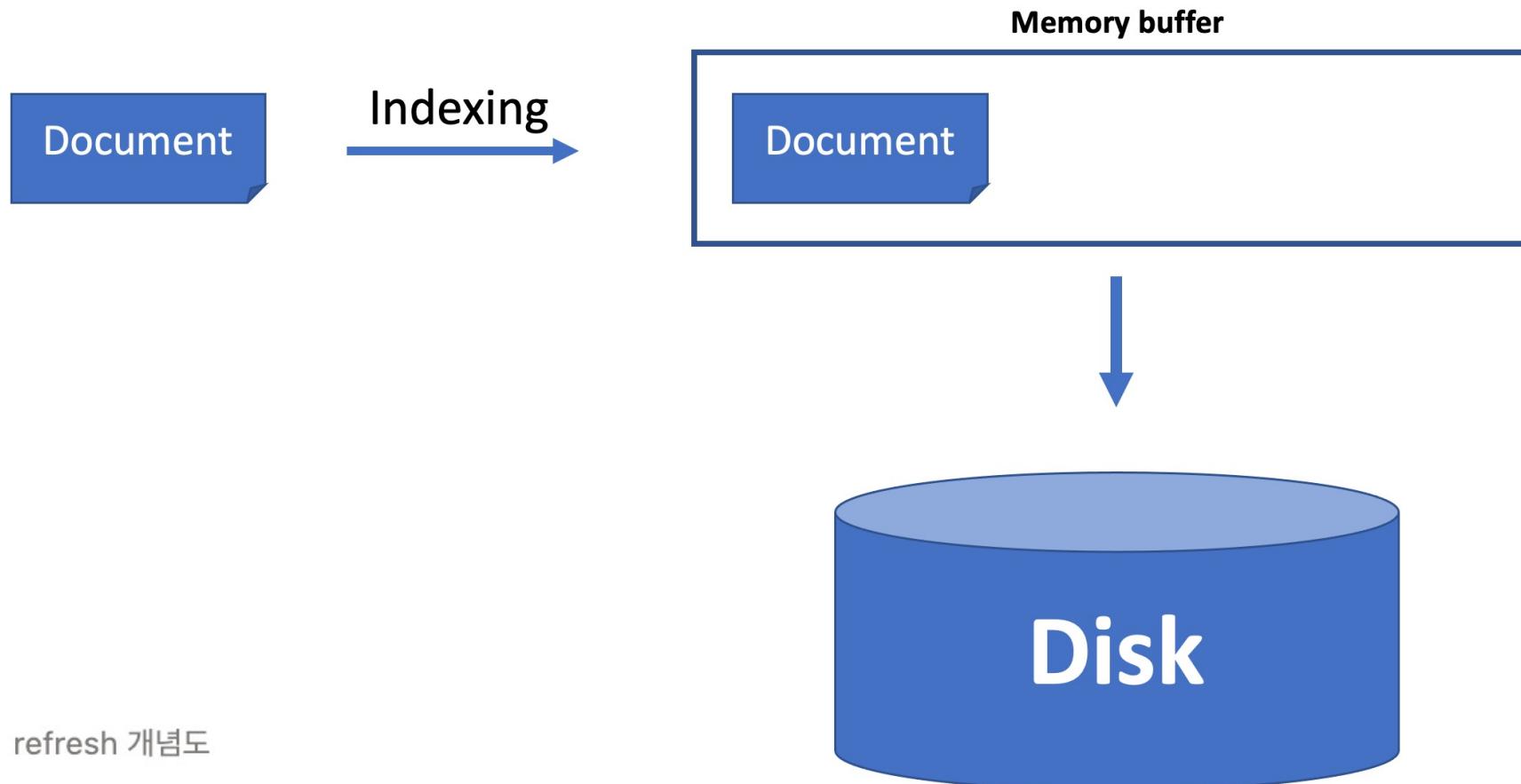
색인된 문서는 이렇게 세그먼트 단위로 저장되어야 검색이 가능해지며,

이런 일련의 작업들을 **refresh** 라고 한다.

이 **refresh** 를 얼마나 주기적으로 할 것인지를 결정하는 값이 **refresh_interval** 이다.

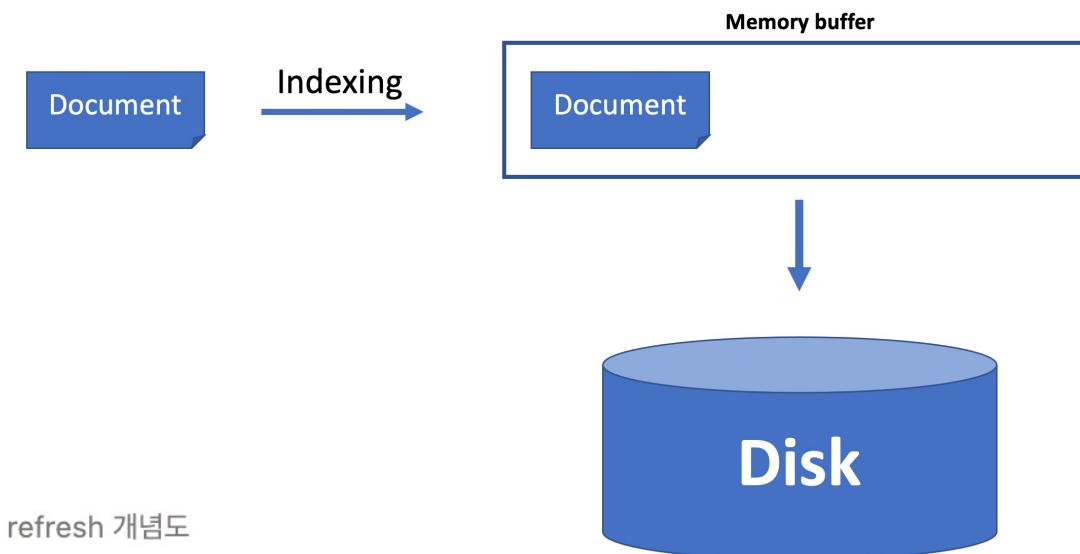
01 색인 성능 최적화

2. refresh_interval 변경하기



01 색인 성능 최적화

2. refresh_interval 변경하기



ElasticSearch가 '준 실시간 검색 엔진' 이라 불리는 것은

이 **refresh_interval** 이 1초로 기본값으로 설정되어 있으므로

문서가 색인되고 1초 후에 검색이 가능하기 때문이다.

01 색인 성능 최적화

2. refresh_interval 변경하기

- 만약 실시간 검색 엔진으로 사용하고자 한다면 **refresh_interval** 을 **1초**로 설정해야 한다.
- 하지만 대용량의 로그를 수집하는 것이 주된 목적이고 색인한 로그를 당장 검색할 필요가 없다면 **refresh_interval** 을 충분히 늘려서 색인 성능을 확보할 수 있다.

01 색인 성능 최적화

2. refresh_interval 변경하기

```
PUT http://localhost:9200/_settings?pretty
{
  "index.refresh_interval" : "2h"
}
```

refresh_interval 설정 요청

01 색인 성능 최적화

03 bulk API

01 색인 성능 최적화

3. bulk API

bulk API 는 한 번에 다양의 문서를 색인, 삭제, 수정할 때 사용할 수 있는 API이다.

01 색인 성능 최적화

3. bulk API

ElasticSearch는 하나의 문서를 처리할 때 시스템은 각 처리마다 커넥션 연결/해제를 수행하기 때문에

여러 건의 문서를 처리하면 시스템에 부하가 발생한다.

그러므로 여러 건의 문서를 처리할 때는 **bulk API** 를 통해 문서 색인, 업데이트 등의

작업을 모아서 한 번에 수행하는 것이 좋다.

01 색인 성능 최적화

3. bulk API

bulk API는 인덱스에 수행할 동작들을 **json** 형태로 나열하여 사용하기도 하고,

해당 내용을 **파일**로 저장하여 사용하기도 한다.

01 색인 성능 최적화

3. bulk API

bulk API에서 수행할 수 있는 동작들

bulk API 동작	설명
index	문서 색인. 인덱스에 지정한 문서 아이디가 있을 때는 업데이트
create	문서 색인. 인덱스에 지정한 문서 아이디가 없을 때에만 색인 가능
delete	문서 삭제
update	문서 변경

01 색인 성능 최적화

<http://ndjson.org/>

ndjson Newline Delimited JSON

3. bulk API

```
POST http://localhost:9200/_bulk?pretty
```

```
Content-Type: application/x-ndjson
```

```
// bulk_index에 id가 1인 문서 색인
{ "index": { "_index": "bulk_index", "_type" : "_doc", "_id" : "1" } }
{ "mydoc": "first_doc" }
{ "index": { "_index": "bulk_index", "_type": "_doc", "_id": "2" } }
{ "mydoc": "second doc" }
{ "index": { "_index": "bulk_index", "_type": "_doc", "_id": "3" } }
{ "mydoc": "third doc" }

...
```

json 형태로 나열된 명령들을 bulk API로 실행

- 첫 줄에는 **index** 를 정의하고, 그 다음 줄에는 인덱스의 문서를 입력한다.

01 색인 성능 최적화

3. bulk API

이전에 예제로 수행했던 **bulk API** 처럼 파일로도 수행할 수 있다.

```
{"index":{}}
```

```
{ "title": "ElasticSearch Training Book", "publisher": "insight", "ISBN": ... }
```

```
{"index":{}}
```

```
{ "title" : "Kubernetes: Up and Running", "publisher": "O'Reilly Media, Inc.", "ISBN": ... }
```

sample.json

```
time curl -X POST 'localhost:9200/dynamic/_doc/_bulk?pretty' \
-s -H 'Content-Type:application/x-ndjson' --data-binary @sample.json > /dev/null
```

01 색인 성능 최적화

3. bulk API

| 다수의 문서에 대한 작업을 다수의 API를 호출하여 처리하는 것보다 *bulk API*를 통해 색인 성능을 최적화할 수 있다.

01 색인 성능 최적화

04 그 외의 색인 성능을 확보하는 방법들

01 색인 성능 최적화

4. 그 외의 색인 성능을 확보하는 방법들

문서의 **id** 를 지정하지 않고 색인하는 방법

문서를 색인할 때 **PUT** 메서드로 문서의 **id** 를 지정하여 색인할 수 있고,

POST 메서드로 ElasticSearch가 문서의 **id** 를 임의로 생성하게 하여 색인할 수도 있다.

01 색인 성능 최적화

4. 그 외의 색인 성능을 확보하는 방법들

문서의 **id** 를 지정하지 않고 색인하는 방법

```
PUT http://localhost:9200/include_id/_doc/1?pretty
{
    "comment" : "It's a document by Put Method"
}
```

문서의 id를 지정하여 색인

```
{
    "_index": "include_id",
    "_type": "_doc",
    "_id": "1",      // id가 1로 지정되어 있다.
    ...
}
```

응답

01 색인 성능 최적화

4. 그 외의 색인 성능을 확보하는 방법들

문서의 **id** 를 지정하지 않고 색인하는 방법

```
POST http://localhost:9200/include_id/_doc?pretty
{
  "comment" : "It's a document"
}
```

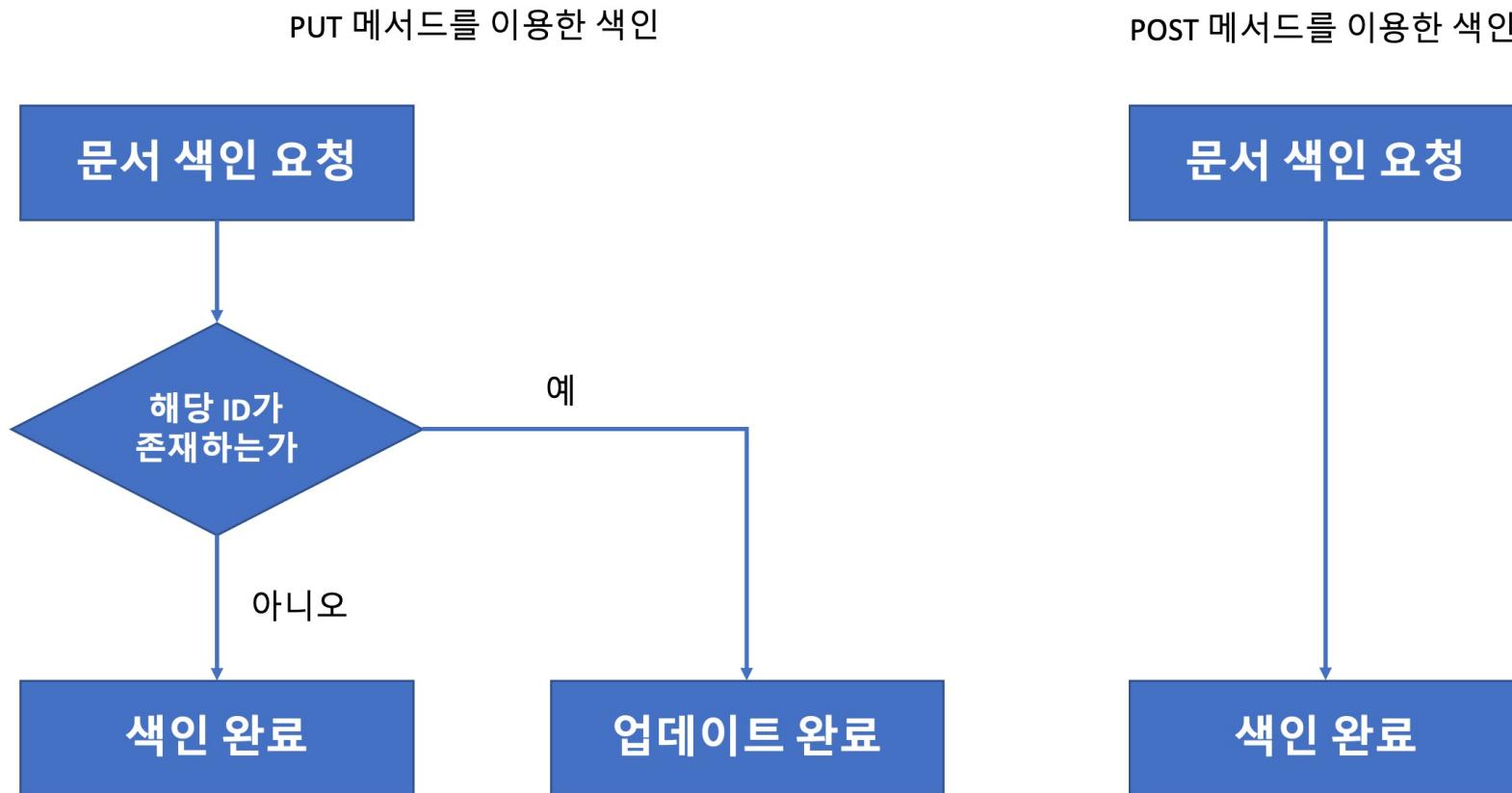
문서의 id를 지정하지 않은 색인

```
{
  "_index": "include_id",
  "_type": "_doc",
  "_id": "5h1ic38BlveLyhKJk0WN", // 임의 id 부여
  ...
}
```

응답

01 색인 성능 최적화

4. 그 외의 색인 성능을 확보하는 방법들



01 색인 성능 최적화

4. 그 외의 색인 성능을 확보하는 방법들

레플리카 샤드 갯수를 0으로 설정

문서 색인이 요청되면 프라이머리 샤드가 완전히 색인이 완료된 이후에 레플리카 샤드를 복제한다.

이렇게 레플리카 샤드의 복제 수행까지 끝낸 다음 클라이언트에게 수행한 작업에 대한 결과를 리턴한다.

하지만 레플리카 샤드가 없다면 자신이 가지고 있는 샤드만 색인한 후 색인 작업을 마무리한다.

01 색인 성능 최적화

4. 그 외의 색인 성능을 확보하는 방법들

레플리카 샤드 갯수를 0으로 설정

즉, 레플리카 샤드가 없다면 전체적인 색인 성능이 향상된다.

데이터가 유실되어 상관없는 경우에는 레플리카 샤드 없이 운영하기도 한다.

01 색인 성능 최적화

1. 정적 매핑은 필요한 필드들만 사용하기 때문에 성능 향상에 도움이 된다.
2. `refresh_interval` 을 적절히 설정하여 디스크 I/O의 빈도수를 줄이면 색인 리소스를 절약할 수 있다.
3. 문서 id 지정이 필요하지 않다면 `POST` 메서드를 이용해 색인하는 것이 PUT 메서드보다 성능 향상에 도움이 된다.
4. 레플리카 샤드가 꼭 필요한 상황이 아니라면 `프라이머리 샤드` 만을 이용해서 운영하는 것도 성능 향상에 도움이 된다.

00 목차

02 검색 성능 최적화

- 01 Elasticsearch 캐시의 종류와 특성
- 02 검색 쿼리 튜닝하기
- 03 샤프 배치 결정하기
- 04 forcemerge API
- 05 그 외의 검색 성능을 확보하는 방법들

02 검색 성능 최적화

01 ElasticSearch 캐시의 종류와 특성

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성

ElasticSearch로 요청되는 **검색 쿼리** 는 빠른 응답을 주기 위해 해당 **쿼리의 결과를 메모리에 저장한다.**

결과를 메모리에 저장해 두는 것을 **메모리 캐싱** 이라고 하며 이때 사용하는 메모리 영역을 **캐시 메모리** 라고 한다.

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성

다음은 ElasticSearch에서 제공하는 대표적인 캐시 영역의 종류이다.

캐시 영역	설명
Node query cache	쿼리에 의해 각 노드에 캐싱되는 영역
Shard request cache	쿼리에 의해 각 샤드에 캐싱되는 영역
Field data cache	쿼리에 의해 필드를 대상으로 각 노드에 캐싱되는 영역

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성

1.1. Node Query Cache

Node Query Cache는 **filter context**에 의해 검색된 문서의 결과가 캐싱되는 영역이다.



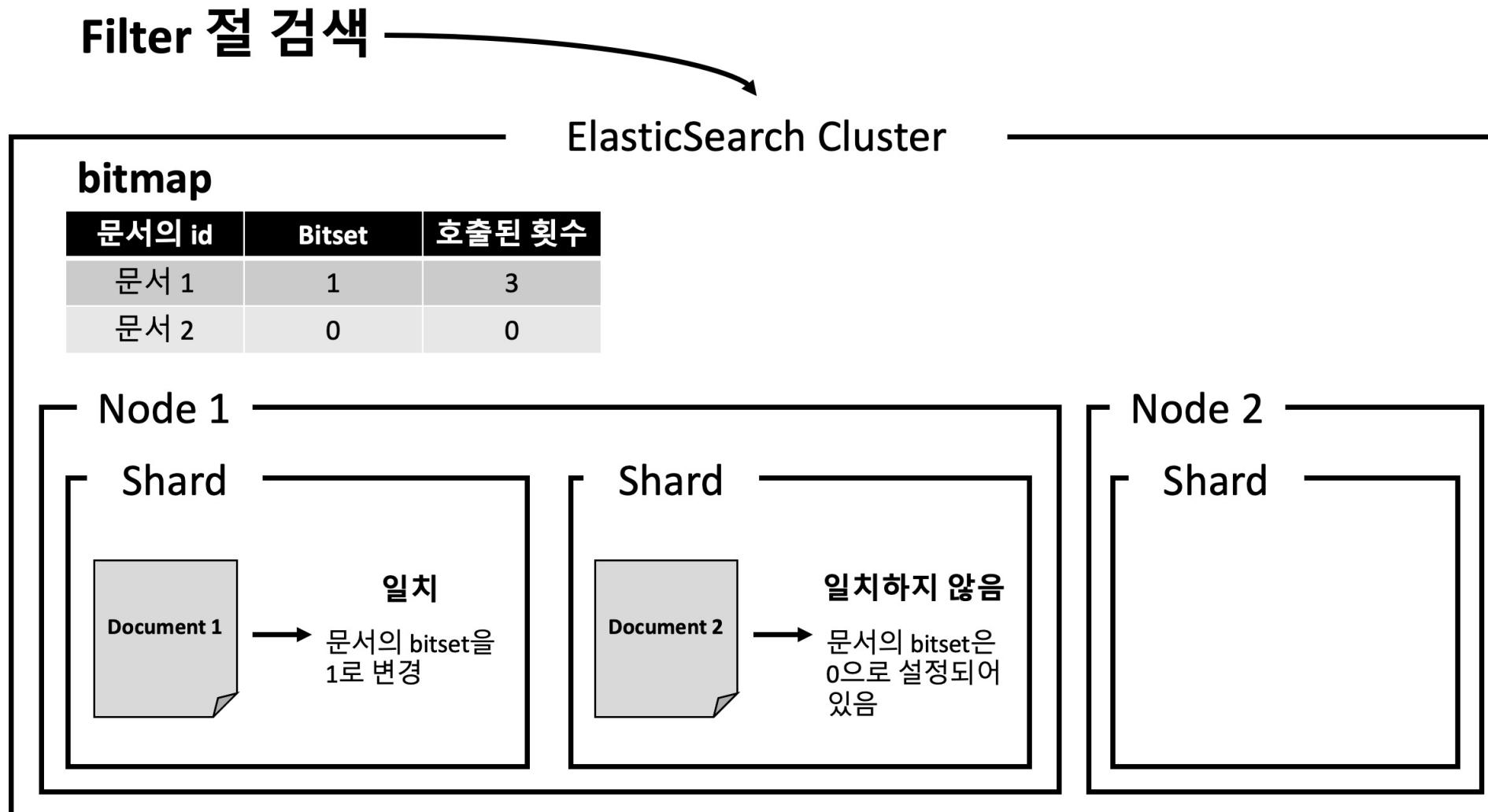
Filter Context (Term Level Query)

문서에 대한 필터링에 사용되는 쿼리이다.

(ex. 일치하는 단어가 있는지, 패턴에 해당하는 값이 있는지, ...)

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.1. Node Query Cache



02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.1. Node Query Cache

예제를 통해 filter context 쿼리의 결과가 어떻게 캐싱되는지 확인해 보자.

```
GET http://localhost:9200/_cat/nodes?v&h=name,qcm&pretty
```

사용된 Query Cache Memory의 용량 조회 요청

name	qcm
aa30a6257d9a	0b

사용된 Query Cache Memory의 용량 조회 응답

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.1. Node Query Cache

예제를 통해 filter context 쿼리의 결과가 어떻게 캐싱되는지 확인해 보자.

```
http://localhost:9200/_cat/segments/static?v&h=index,shard,segment,docs.count
```

특정 인덱스의 세그먼트 조회 요청

	index	shard	segment	docs.count
static	0	_0		100000

응답

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.1. Node Query Cache

filter context로 구성된 검색 쿼리를 요청해보자.

GET http://localhost:9200/static/_search?pretty

```
{  
  "from": 0,  
  "size": 10000,  
  "query": {  
    "bool": {  
      "filter": {  
        "range": {  
          "release_date": {  
            "gte": "2017/09/01"  
          }  
        }  
      }  
    }  
  }  
}
```

filter context로 구성된 검색 쿼리 요청

```
{  
  // 최종 요청이 리턴되는 데 걸린 시간 (118ms)  
  "took": 118,  
  ...  
  "hits": [  
    {  
      ...  
      "_source": {  
        "title": "ElasticSearch Training Book",  
        "publisher": "insight",  
        "ISBN": "9788966264849",  
        "release_date": "2020/09/30",  
        ...  
      }  
    }  
  ]  
}
```

응답

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.1. Node Query Cache

```
GET http://localhost:9200/_cat/nodes?v&h=name,qcm&pretty
```

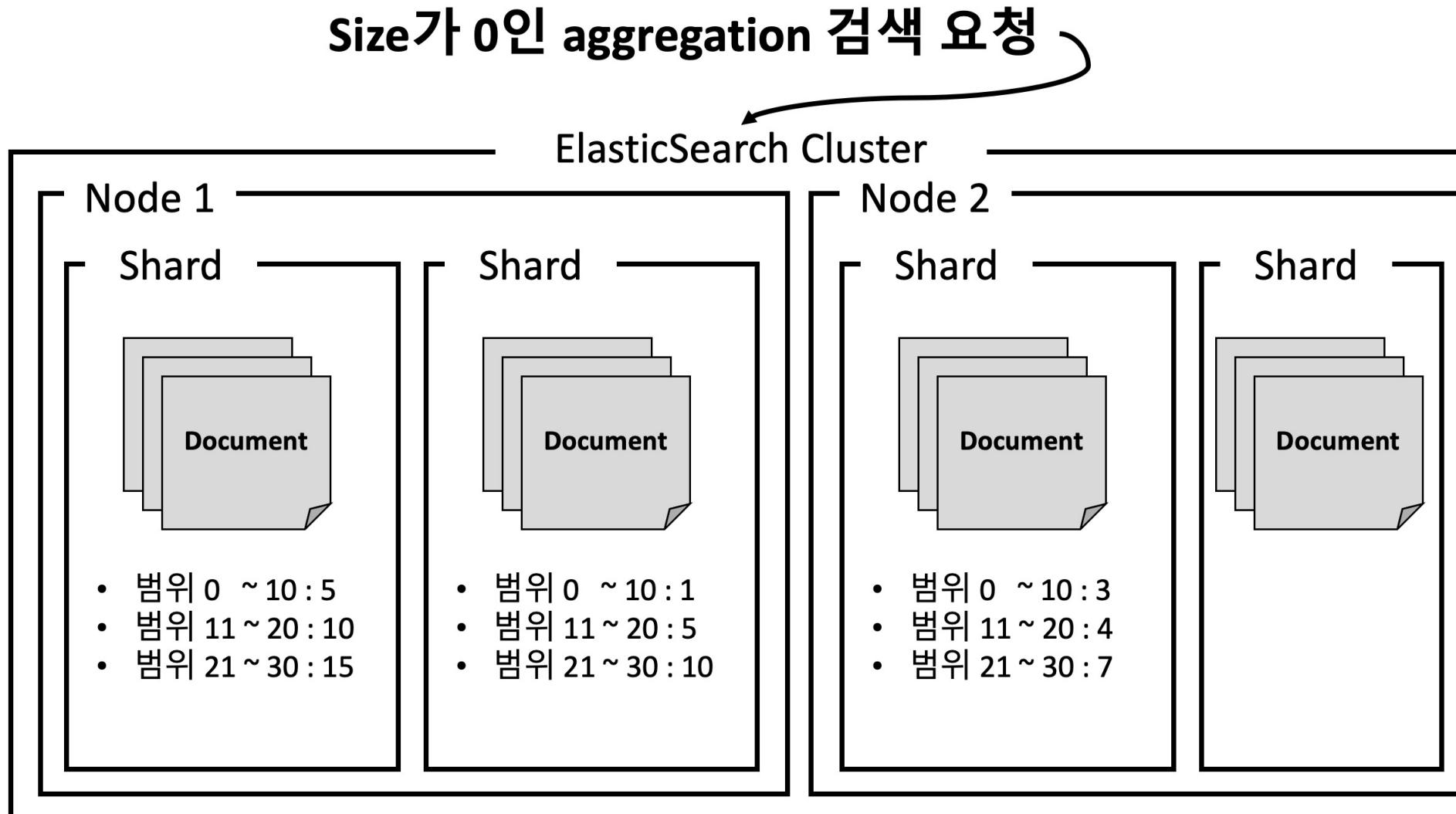
사용된 Query Cache Memory의 용량 조회 요청

name	qcm
aa30a6257d9a	13.3kb

- 즉, 자주 조회되는 문서에 대해 캐싱하여 성능을 향상시키는 것을 확인할 수 있다.

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.2. Shard Request Cache



02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.2. Shard Request Cache

다만 이 영역은 샤드에 **refresh** 동작을 수행하면 캐싱된 내용이 사라진다.

즉, 문서 색인이나 업데이트를 한 이후 **refresh** 를 통해 샤드의 내용이 변경되면 캐싱 결과가 초기화된다.

따라서 색인이 자주 일어나는 인덱스에는 비효율적이다.

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.2. Shard Request Cache

```
GET http://localhost:9200/_cat/nodes?v&h=name, rcm&pretty
```

Shard Request Cache 현황 조회 요청

name	rcm
aa30a6257d9a	0b

응답

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.2. Shard Request Cache

```
GET http://localhost:9200/static/_search?pretty
{
  "size": 0,
  "aggs": {
    "titleaggs": {
      "terms": {
        "field": "title"
      }
    }
  }
}
```

동일한 제목이 몇개 인지 집계

```
{
  // 62ms 소요
  "took": 62,
  ...
  "aggregations": {
    "titleaggs": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": "Cloud Native Java",
          "doc_count": 10000
        },
        ...
      ]
    }
  }
}
```

응답

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.2. Shard Request Cache

```
GET http://localhost:9200/_cat/nodes?v&h=name, rcm&pretty
```

Shard Request Cache 현황 조회 요청

name	rcm
aa30a6257d9a	744b

응답

- 집계 결과에 대해 캐싱된 것을 확인할 수 있다.

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.2. Shard Request Cache

```
GET http://localhost:9200/static/_search?pretty
```

```
{  
    "size": 0,  
    "aggs": {  
        "titleaggs": {  
            "terms": {  
                "field": "title"  
            }  
        }  
    }  
}
```

동일한 제목이 몇개 인지 집계

```
{  
    "took": 4,           // 4ms  
    "timed_out": false,  
    "_shards": {  
        ...  
    }  
}
```

응답

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.2. Shard Request Cache

```
POST http://localhost:9200/static/_doc  
  
{  
  "ISBN": "97889662641212",  
  "title": "Shared Request Cache qweqeqewqe",  
  "publisher": "sangmin",  
  "release_date": "2022/03/12",  
  "description": "테스트"  
}
```

문서 색인 요청

```
GET http://localhost:9200/_cat/nodes?v&h=name,rcm&pretty
```

Shard Request Cache 현황 조회

name	rcm
aa30a6257d9a	0b

응답

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.3. Field Data Cache

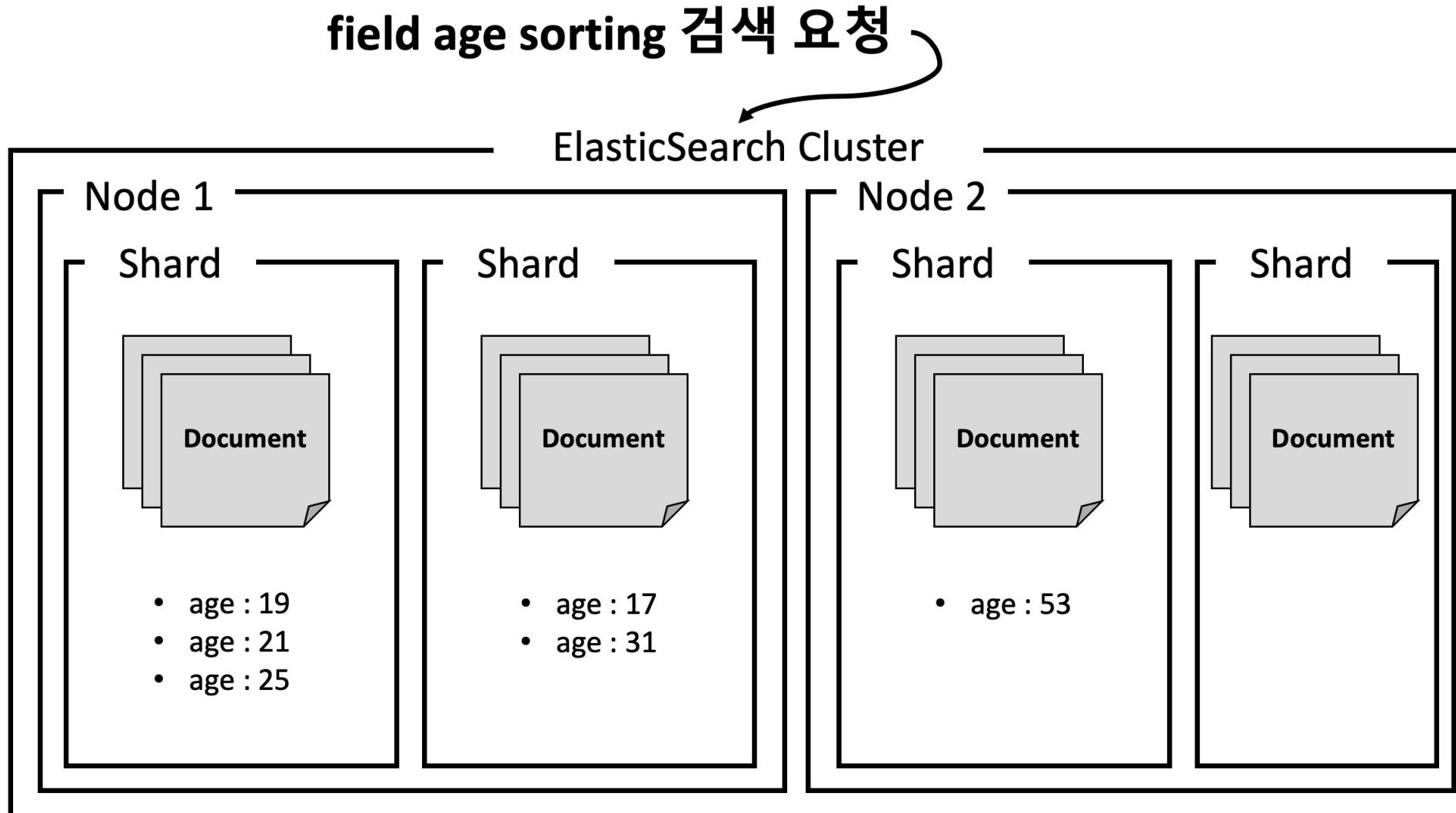
Field Data Cache 는 인덱스를 구성하는 필드에 대한 캐싱이다.

이 영역은 주로 검색 결과를 정렬하거나 집계 쿼리를 수행할 때 지정한 **필드만** 을 대상으로

해당 필드의 모든 데이터를 메모리에 저장하는 캐시 영역이다.

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.3. Field Data Cache



02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.3. Field Data Cache

예제를 통해 Field Data Cache 메모리 사용 현황을 살펴보자.

```
GET http://localhost:9200/_cat/nodes?v&h=name,fm&pretty
```

Field Data Cache 현황 조회

```
name          fm  
aa30a6257d9a 0b
```

응답

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.3. Field Data Cache

예제를 통해 Field Data Cache 메모리 사용 현황을 살펴보자.

```
GET http://localhost:9200/static/_search
{
  "aggs": {
    "isbn_count": {
      "terms": {
        "field": "ISBN"
      }
    }
  }
}
```

```
{
  "took": 47,
  ...
}
```

- 47ms 소요

02 검색 성능 최적화

1. ElasticSearch 캐시의 종류와 특성 1.3. Field Data Cache

예제를 통해 Field Data Cache 메모리 사용 현황을 살펴보자.

```
GET http://localhost:9200/_cat/nodes?v&h=name, fm&pretty
```

Field Data Cache 현황 조회

```
name          fm  
aa30a6257d9a 832b
```

응답

02 검색 성능 최적화

02 검색 쿼리 튜닝하기

02 검색 성능 최적화

2. 검색 쿼리 튜닝하기

검색 성능을 떨어뜨리는 요인 중 하나는 너무 많은 필드를 조회하는 것이다.

이렇게 많은 필드를 하나의 필드로 모아서 검색할 수 있는 기능으로 `copy_to` 가 제공된다.

02 검색 성능 최적화

2. 검색 쿼리 튜닝하기

예제를 통해 자세히 살펴보자.

```
PUT http://localhost:9200/copyto_index  
  
{  
  "mappings": {  
    "properties": {  
      "first_name": {  
        "type" : "text"  
      },  
      "last_name" : {  
        "type" : "text"  
      }  
    }  
  }  
}
```

색인 생성

```
{  
  "acknowledged": true,  
  "shards_acknowledged": true,  
  "index": "copyto_index"  
}
```

응답

02 검색 성능 최적화

2. 검색 쿼리 튜닝하기

예제를 통해 자세히 살펴보자.

```
POST http://localhost:9200/copyto_index/_doc
{
  "first_name": "alden",
  "last_name": "kang"
}
```

문서 생성

```
{
  "_index": "copyto_index",
  "_type": "_doc",
  "_id": "pIDqgX8BQ5Vx4a2Uu7sl",
  ...
}
```

응답

```
GET http://localhost:9200/copyto_index/_search?pretty
{
  "query": {
    "bool": {
      "must": [
        { "match": { "first_name": "alden" } },
        { "match": { "last_name": "kang" } }
      ]
    }
  }
}
```

생성한 문서 조회

02 검색 성능 최적화

2. 검색 쿼리 튜닝하기

예제를 통해 자세히 살펴보자.

```
GET http://localhost:9200/copyto_index/_search?pretty
{
  "query": {
    "bool": {
      "must": [
        { "match": { "first_name": "alden" } },
        { "match": { "last_name": "kang" } }
      ]
    }
  }
}
```

생성한 문서 조회

```
"took": 6,
"timed_out": false,
"_shards": {
  ...
},
"hits": {
  ...
  "max_score": 0.5753642,
  "hits": [
    {
      ...
      "_source": {
        "first_name": "alden",
        "last_name": "kang"
      }
    }
  ]
}
```

`first_name` 필드와 `last_name` 필드를 대상으로 검색하기 위해 `match` 쿼리를 두 번 사용했다.

02 검색 성능 최적화

2. 검색 쿼리 튜닝하기

이처럼 두 개의 필드를 항상 사용해야 한다면 **copy_to** 를 이용하여 하나의 필드에서 검색하는 것이 검색 성능에 더 좋다.

02 검색 성능 최적화

2. 검색 쿼리 튜닝하기

Copy-index

“last_name” : “button , “first_name” : “Benjamin”

Copy_to

“full_name” : “**button Benjamin**”

02 검색 성능 최적화

2. 검색 쿼리 튜닝하기

```
PUT http://localhost:9200/copyto_index  
  
{  
  "mappings": {  
    "properties": {  
      "first_name": {  
        "type": "text",  
        "copy_to": "full_name"  
      },  
      "last_name": {  
        "type": "text",  
        "copy_to": "full_name"  
      },  
      "full_name": {  
        "type": "text"  
      }  
    }  
  }  
}
```

copy_to 필드를 활용하여 단일 필드가 적용된 copyto_index 생성

```
POST http://localhost:9200/copyto_index/_doc  
  
{  
  "first_name": "alden",  
  "last_name": "kang"  
}
```

문서 생성

```
GET http://localhost:9200/copyto_index/_search?pretty  
  
{  
  "query": {  
    "match": {  
      "full_name" : "alden kang"  
    }  
  }  
}
```

단일 필드로 문서 조회

02 검색 성능 최적화

2. 검색 쿼리 튜닝하기

```
GET http://localhost:9200/copyto_index/_search?pretty
{
  "query": {
    "match": {
      "full_name" : "alden kang"
    }
  }
}
```

단일 필드로 문서 조회

```
{
  "took": 1154,
  ...
  "_source": {
    "first_name": "alden",
    "last_name": "kang"
  }
}
```

응답

- 이처럼 `copy_to` 로 지정된 필드를 `bool` 쿼리를 사용하여 단일 쿼리로 조회할 수 있다.
- 많은 필드를 대상으로 검색해야 한다면 `copy_to` 기능을 최대한 활용하도록 하자.

02 검색 성능 최적화

03 색드 배치 결정하기

02 검색 성능 최적화

3. 샤프 배치 결정하기

ElasticSearch는 프라이머리 샤프의 개수를 한번 설정하면 변경할 수 없기 때문에 처음 샤프 개수를 설정할 때 신중하게 설정해야 한다.

샤프 배치를 잘못했을 경우 겪는 문제 중 하나는 클러스터 내의 데이터 노드 간 볼륨 사용량이 불균형해지는 문제이다.

02 검색 성능 최적화

3. 색드 배치 결정하기

Cluster	
<i>Node</i>	<i>Index</i>
node1	Shard 0, Shard 3
node2	Shard 1
node3	Shard 2

02 검색 성능 최적화

3. 샤크 배치 결정하기

Cluster	
<i>Node</i>	<i>Index</i>
node1	Shard 0
node2	Shard 1
node3	

02 검색 성능 최적화

3. 샤프 배치 결정하기

즉, 이러한 문제를 해결하기 위해서는 노드의 개수에 적절하게 인덱스의 샤프 개수를 설정해야 클러스터 성능에 유리하다.

그렇다고 해서 노드의 개수와 샤프의 개수를 동일하게 설정해도 문제가 발생할 수 있다.

02 검색 성능 최적화

3. 샤드 배치 결정하기

노드의 개수와 샤드의 개수를 동일하게 설정한 상태에서 노드를 증설한 경우를 살펴보자.

Cluster	
<i>Node</i>	<i>Index</i>
node1	Shard 0
node2	Shard 1
node3	Shard 2



Cluster	
<i>Node</i>	<i>Index</i>
node1	Shard 0
node2	Shard 1
node3	Shard 2
node4	

02 검색 성능 최적화

3. 샤드 배치 결정하기

최초 구성한 노드의 개수 와 증설된 이후 노드의 개수 의 최소 공배수로 샤드의 개수를 설정하면 이전과 같은 문제를 모두 예방할 수 있다.

Cluster	
<i>Node</i>	<i>Index</i>
node1	Shard 0, 3, 6, 9
node2	Shard 1, 4, 7, 10
node3	Shard 2, 5, 8, 11



Cluster	
<i>Node</i>	<i>Index</i>
node1	Shard 0, 6, 9
node2	Shard 1, 4, 10
node3	Shard 2, 5, 8
node4	Shard 3, 7, 11

02 검색 성능 최적화

3. 샤드 배치 결정하기

추가적으로 성능을 향상시키기 위해서는 데이터 노드에 할당된 샤드들의 정보를 알고 있어야 하는 **마스터 노드**에 관해서도 고려할 점이 있다.

노드나 인덱스, 샤드가 증가할수록 **마스터 노드**는 이 정보들을 관리하기 위해 메모리와 리소스를 점점 더 많이 사용하게 된다.

클러스터 성능이 제대로 나오지 않는다면 마스터 노드의 성능을 확인해 보아야 한다.

02 검색 성능 최적화

3. 샤프 배치 결정하기

ElasticSearch는 이렇게 클러스터 내에 샤프가 너무 많아져서 전체 성능이 저하되는 것을 막기 위해 노드의 샤프의 개수를 제한하는 설정이 있다.

```
PUT http://localhost:9200/_cluster/settings  
  
{  
  "transient": {  
    "cluster.max_shards_per_node" : 2000  
  }  
}
```

클러스터 내에 노드당 샤프 조회 제한 설정

- 노드당 최대 샤프 개수를 2,000개로 설정했다.

02 검색 성능 최적화

04 forcemerge API

02 검색 성능 최적화

4. forcemerge API

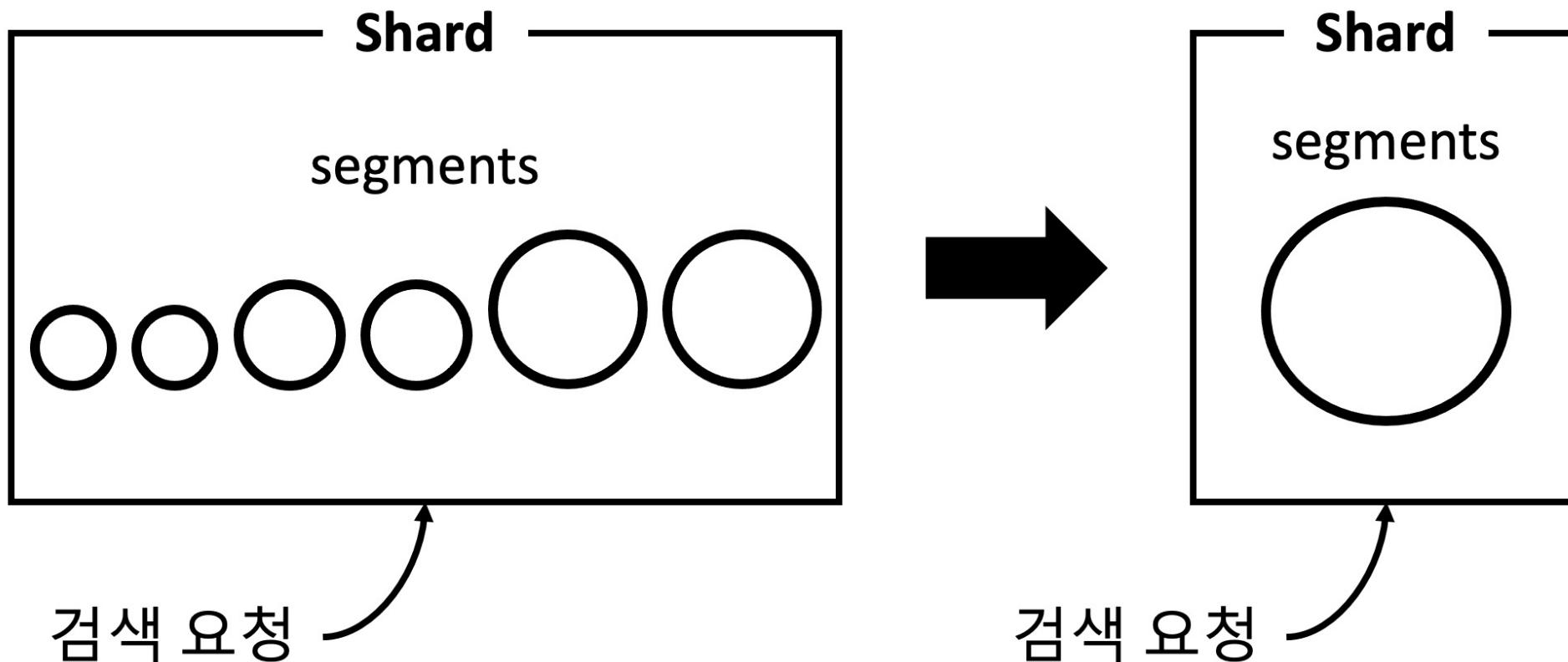
검색 성능을 향상시키기 위해 세그먼트를 강제로 병합시키는 **forcemerge API**에 대해 알아보자.

02 검색 성능 최적화

4. forcemerge API

forcemerge API를 살펴보기 전에 **세그먼트** 를 떠올려 보자.

많은 세그먼트로 구성된 샤드와 하나의 세그먼트로 구성된 샤드의 검색 요청



02 검색 성능 최적화

4. forcemerge API

forcemergetest는 500,000개의 문서가 색인된 상태이다.

```
GET http://localhost:9200/_cat/segments/forcemergetest?v&h=index,shard,prirep,segment,docs,count,size
```

forcemergetest 인덱스의 샤드 및 세그먼트 조회

index	shard	prirep	segment	size
forcemergetest	0	p	_0	357.5kb
forcemergetest	0	p	_1	1.1mb
forcemergetest	0	p	_2	2.3mb
forcemergetest	0	p	_3	2.9mb
forcemergetest	0	p	_4	2.8mb
forcemergetest	0	p	_5	740.8kb
forcemergetest	0	p	_6	29.4mb
forcemergetest	0	p	_7	9.8mb

응답

02 검색 성능 최적화

4. forcemerge API

```
POST http://localhost:9200/_forcemergetest/_forcemerge?max_num_segments=1
```

forcemergetest 인덱스의 세그먼트 병합

- 샤크당 하나의 세그먼트로 병합 요청

02 검색 성능 최적화

4. forcemerge API

```
GET http://localhost:9200/_cat/segments/forcemergetest?v&h=index,shard,prirep,segment,docs,count,size
```

forcemergetest 인덱스의 샤드 및 세그먼트 조회

index	shard	prirep	segment	size
forcemergetest	0	p	_8	49.1mb

응답

02 검색 성능 최적화

4. forcemerge API

즉, 클러스터의 특성에 맞게 샤드 개수를 적절히 설정하고, 샤드 내 세그먼트를 병합할 때 적절한 용량으로 병합하는 것이 중요하다.

02 검색 성능 최적화

05 그 외의 검색 성능을 확보하는 방법들

02 검색 성능 최적화

5. 그 외의 검색 성능을 확보하는 방법들

nested 타입

ElasticSearch에서는 문서를 모델링할 때 가급적이면 간결하게 구성하도록 권고한다.

Parent/Child 구조의 **join** 구성이나 **nested** 타입같이 문서를 처리할 때 문서 간의 연결 관계 처리를 필요로 하는 구성은 권장하지 않는다.

02 검색 성능 최적화

5. 그 외의 검색 성능을 확보하는 방법들

nested 타입

```
PUT http://localhost:9200/nested_index
{
  "mappings": {
    "properties": {
      "name": {
        "type": "keyword"
      },
      "devices": {
        "type": "nested"
      }
    }
  }
}
```

nested datatype으로 구성된 인덱스 생성

```
POST http://localhost:9200/nested_index/_doc
{
  "name": "alden",
  "devices": [
    {
      "device_name": "iPhone11"
    },
    {
      "device_name": "iPad Pro"
    },
    {
      "device_name": "Galaxy Note 10"
    }
  ]
}
```

nested datatype으로 구성된 문서 생성

02 검색 성능 최적화

5. 그 외의 검색 성능을 확보하는 방법들

nested 타입

만약 nested 타입을 사용하지 않았다면 문서 3건을 저장해야 한다.

이렇게 단순히 문서 개수만 봤을 때는 nested 가 빠를 것으로 예상되지만,

실제로는 문서가 점점 많아지면 문서 간의 연결 고리를 고려해야 하는 nested 보다

일반적인 형태로 문서를 여러 개로 모델링하는 쪽이 성능이 좋다.

```
POST http://localhost:9200/nested_index/_doc
```

```
{  
  "name": "alden",  
  "devices": "iPhone11"  
}
```

```
POST http://localhost:9200/nested_index/_doc
```

```
{  
  "name": "alden",  
  "devices": "iPad Pro"  
}
```

```
POST http://localhost:9200/nested_index/_doc
```

```
{  
  "name": "alden",  
  "devices": "Galaxy Note 10"  
}
```

02 검색 성능 최적화

5. 그 외의 검색 성능을 확보하는 방법들

레플리카 샤드

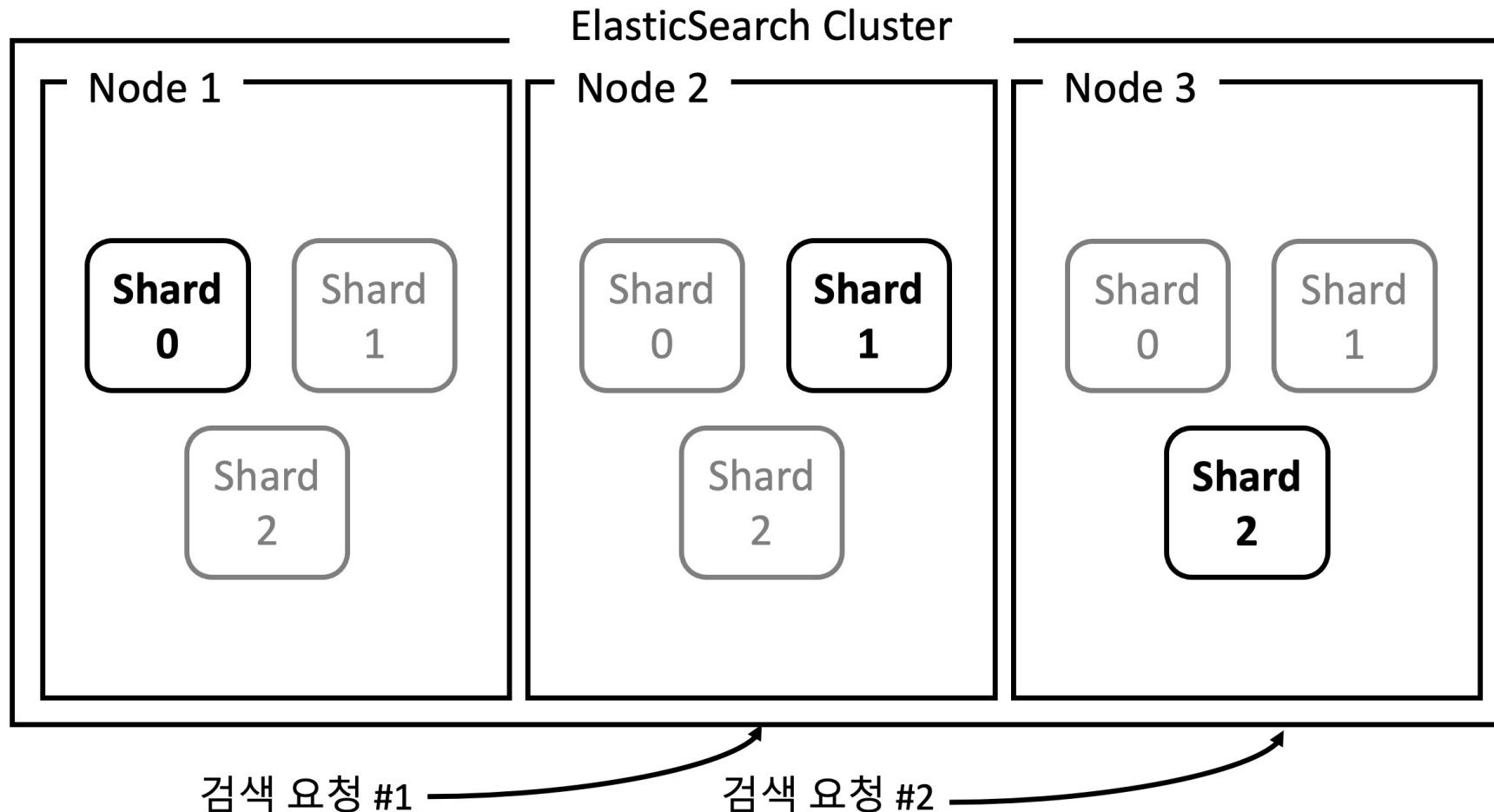
ElasticSearch는 레플리카 샤드를 가능한 한 충분히 둘 것을 권고한다.

레플리카 샤드 는 데이터 안정성을 확보하는 역할을 할 뿐만 아니라, 검색 요청에 대해 많은 샤드들이 결과를 리턴해 주는 역할을 하기 때문이다.

02 검색 성능 최적화

5. 그 외의 검색 성능을 확보하는 방법들

레플리카 샤드



02 검색 성능 최적화

- **Node Query Cache** : 쿼리 결과를 각 노드에 캐싱하는 영역
- **Shard Request Cache** : 쿼리 결과 중 집계 데이터에 관한 결과를 각 샤드에 캐싱하는 영역
- **Field Data Cache** : 쿼리의 대상이 되는 필드의 데이터들을 캐싱하는 영역

02 검색 성능 최적화

- 문서를 검색할 대상 필드를 줄이거나, 쿼리의 종류를 바꾸는 것만으로도 성능을 향상시킬 수 있다.
- **forcemerge API** 를 활용해 세그먼트를 강제로 병합하여 검색 성능을 향상시킬 수 있다.
- 꼭 필요한 상황이 아니면 **nested** 타입을 사용하지 말아야 한다.
- 레플리카 샤드를 추가하여 검색 성능을 향상시킬 수 있다.

Thank You