실무 예제로 배우는

Elasticsearch 검색엔진 활용편

정호욱 지음

한빛미디어 Hanbit Media, Inc.

실무 예제로 배우는

Elasticsearch 검색엔진 활용편

정호욱 지음

한빛미디어 Hanbit Media, Inc.

실무 예제로 배우는 **Elasticsearch 검색엔진** 활용편

초판발행 2015년 3월 9일

지은이 정호욱 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-6848-742-2 15000 / **정가** 15,000원

총괄 배용석 / **책임편집** 김창수 / **기획·편집** 정지연
디자인 표지/내지 여동일, 조판 최송실
마케팅 박상용 / **영업** 김형진, 김진불, 조유미

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanbit.co.kr / **이메일** ask@hanbit.co.kr

Published by HANBIT Media, Inc. Printed in Korea

저자 소개

지은이_ **정호욱**

지난 13년 동안 야후코리아, NHN Technology, 삼성전자에서 커뮤니티, 소셜 검색, 광고 검색 관련 서비스를 개발해 오면서 검색엔진을 활용한 다양한 프로젝트를 수행하였다. 현재 빅데이터 전문 기업인 그루터Gruter에서 오픈소스 기반 검색엔진 개발자로 근무하고 있다. Elasticsearch 기술에 대한 정보와 경험을 현재 개인 블로그(http://jjeong.tistory.com)를 통해 공유하고 있다.

검색엔진은 모든 서비스의 기본이 되는 핵심 요소입니다. 우리가 사용하는 모든 서비스에는 검색 기능이 포함되어 있습니다. 하지만 검색엔진 관련 기술은 일반 사용자가 접근하기에는 너무 어려운 기술로 남아 있습니다. 루씬Lucene이라는 오픈소스 검색라이브러리가 진입 장벽을 많이 낮추기는 했지만, 서비스에 적용하기에는 개발자가 직접 구현해야 하는 기능이 너무 많고 관리와 유지보수가 어렵다는 문제가 있었습니다.

하지만 이런 문제점은 Elasticsearch라는 오픈소스 검색엔진이 나오면서 사라졌고 전문적인 검색엔진 및 서비스 개발자가 아니더라도 누구나 쉽게 검색 서비스를 만들 수 있게 되었습니다.

비싼 라이선스 비용을 내고 검색 품질과 기능을 커스터마이징하기 어려운 벤더 중심의 검색엔진을 사용하고 있다면 Elasticsearch로 꼭 바꾸길 추천합니다. 아직 국내에는 Elasticsearch 사용자층이 넓지 않습니다. 이 책은 Elasticsearch에 관심은 있으나 어디서부터 시작해야 할지 모르는 사용자와 검색을 모르는 사용자가 쉽게 서비스를 만들 수 있도록 도움을 주고자 집필하였습니다.

끝으로 이 책을 집필하는 데 많은 도움을 주신 그루터 권영길 대표님 그리고 이 책이 세상에 빛을 볼 수 있도록 많은 도움을 주신 한빛미디어 김창수 님, 정지연 님, 이중민 님께 감사의 말을 전합니다.

집필을 마치며

정호욱

대상 독자 및 참고사항

이 책은 검색엔진을 이용한 다양한 기술과의 접목과 활용, 사용자 정의 기능을 구현해서 적용할 수 있는 플러그인 구현 방법까지 Elasticsearch를 적극적으로 활용할 수 있는 방법을 보여줍니다. 또한, 기본적인 성능 최적화 방법과 가이드를 제공하여 대용량 트래픽의 처리와 안정성을 확보하는 데 도움을 줄 수 있도록 구성되어 있습니다.

이 책을 읽으시려면 루씬에 대한 기본 지식이 필요합니다. 또한, 설치와 구성 등 기본적인 내용은 이 책에서 다루지 않으므로 이 책의 전작인 『실무 예제로 배우는 Elasticsearch 검색엔진(기본편)』(한빛미디어, 2014)을 읽어 보시길 권합니다. 사용하는 용어나 기술에 대한 기본 지식이 없을 경우 이해하는 데 어려움이 있을 수 있습니다.

이 도서의 예제 소스 코드는 다음에서 내려받을 수 있습니다.

- https://github.com/HowookJeong?tab=repositories

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾기도 쉽지 않습니다. 또한, 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1 eBook Only – 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다

500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 좀 더 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한, 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2 무료로 업데이트되는 전자책 전용 서비스입니다

종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3 독자의 편의를 위해 DRM-Free로 제공합니다

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 어려운 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

차례

실무 예제로 배우는

Elasticsearch 검색엔진

활용편

『실무 예제로 배우는 Elasticsearch 검색엔진 〈기본편〉』에서는 Elasticsearch의 기본 개념과 설치 방법, 검색서비스 구성까지 살펴봤습니다. 이번 〈활용편〉에서는 〈기본편〉에서 다루지 못한 확장 기능과 다양한 서비스의 활용 방법, Elasticsearch의 성능 최적화 방법을 알아보겠습니다.

chapter 1

검색 기능 확장

1.1 자동 완성

자동 완성Auto Completion 기능은 검색 서비스에서 가장 많이 사용하는 기능의 하나로, 사용자가 입력하는 검색 쿼리를 실시간으로 입력받아 문장을 완성합니다. 이 기능은 키워드 추천이나 오타 교정 등에 활용할 수 있습니다.

검색 서비스에서 자동 완성 기능은 전방 일치, 부분 일치, 후방 일치 기능을 제공합니다. 전방 일치는 입력한 질의가 문장의 앞부분에서 매칭이 이루어지는 것을 의미하고, 부분 일치는 입력한 질의가 문장의 중간에서 매칭되는 것을 의미합니다. 그리고 후방 일치는 전방 일치와 반대로 문장의 뒷부분에서 매칭됩니다. 전방/부분/후방 일치로 매칭된다는 것은 형태소 분석으로 추출된 토큰Token 단위의 색인어(Term)가 일치되는 것을 의미합니다.

Elasticsearch에서는 이런 자동 완성 기능을 구현하기 위해 Prefix 쿼리, Suggester, Analyzer(ngram, edge ngram)를 이용합니다. Prefix 쿼리는 전방 일치 기능을 구현할 때 손쉽게 적용할 수 있는데, 적용을 위해서는 필드Field[01]의 인덱스Index[02] 속성이 not_analyzed가 되어야 합니다. Suggester는 Term, Phrase, Completion, Context Suggester의 4가지 기능이 있으며 아직 개

01 RDBMS에서 테이블의 column에 해당한다.

02 데이터를 저장하기 위한 장소로, RDBMS의 데이터베이스와 유사하다.

발 중입니다. 이 기능들은 유사한 색인어를 찾아 추천해 줍니다. analyzer 중 ngram과 edge ngram을 이용하는 것은 자동 완성용 필드의 텍스트를 미리 분석하여 색인하는 방법입니다.

다음 그림은 '가', '나', '다', '라'를 각각의 색인어라고 가정했을 때 전방/부분/후방 일치에 대한 예입니다. 각 그림의 앞 항목은 입력한 질의어고, 뒤 항목은 분석되어 저장된 단어가 매칭된 모습입니다.

그림 1-1 전방 일치

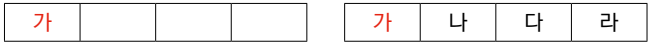


그림 1-2 부분 일치

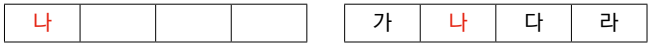
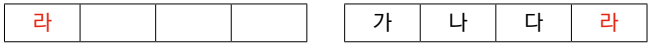


그림 1-3 후방 일치



1.1.1 자동 완성 Analyzer

자동 완성용 indice[03]를 생성할 때 analyzer의 설정을 살펴보겠습니다. 자동 완성 기능을 사용하려면 자동 완성 질의 필드에 용도에 맞춰 tokenizer를 구성해야 합니다.

> NOTE Tokenizer
>
> 색인 과정에서 루씬에 전달된 일반 텍스트는 Tokenization이라는 과정을 거치게 됩니다. 이는 Token이라는 인덱스의 작은 요소로 입력 텍스트를 검색할 수 있도록 처리하는 것을 말합니다. 이러한 작업은 단순히 일반 텍스트를 분리하는 것뿐만 아니라 텍스트를 제거, 변형, 임의의 패턴 매칭, 필터링, 텍스트 정규화 그리고 동의어 확장까지 다양한 처리를 하게 되는데, 이를 담당하는 요소가 Tokenizer입니다.

03 Index는 포괄적인 의미의 색인 또는 색인 파일이고, Indice는 Elasticsearch 내에서 물리적으로 사용되는 색인 또는 색인 파일이라고 보면 된다. Indice는 기존 검색엔진의 collection과 같다.

[Analyzer 설정]

```
"analyzer" : {
    "ngram_analyzer" : {
        "type" : "custom",
        "tokenizer" : "ngram_tokenizer",
        "filter" : ["lowercase", "trim"]
    },
    "edge_ngram_analyzer" : {
        "type" : "custom",
        "tokenizer" : "edge_ngram_tokenizer",
        "filter" : ["lowercase", "trim"]
    },
    "edge_ngram_analyzer_back" : {
        "type" : "custom",
        "tokenizer" : "edge_ngram_tokenizer",
        "filter" : ["lowercase", "trim", "edge_ngram_filter_back"]
    }
}
```

ngram은 음절 단위로 색인어를 생성하는 방식으로 재현율은 높으나 정확도는 떨어집니다. ngram_tokenizer, lowercase 필터, trim 필터로 구성하고, 첫 음절을 기준으로 max_gram에서 지정한 최대 길이만큼 색인어를 생성합니다.

[ngram 색인 결과]

```
min_gram: 1
max_gram: 5
text: 실무 예제로 배우는 검색엔진
terms: ["실", "실무", "무", "예", "예제", "예제로", "제", "제로", "로", "배", "배우", "배우는", "
우", "우는", "는", "검", "검색", "검색엔", "검색엔진", "색", "색엔", "색엔진", "엔", "엔진", "진"]
```

edge ngram은 ngram과 매우 유사한데, min_gram 크기부터 max_gram 크기까지 지정한 tokenizer의 특성에 맞춰 각 색인어에 대한 ngram 색인어를 생성하는 방식입니다. edge_ngram_tokenizer, lowercase 필터, trim 필터로 구성합니다.

[edge ngram 색인 결과]

```
min_gram: 1
max_gram: 5
text: 실무 예제로 배우는 검색엔진
terms: ["실", "실무", "예", "예제", "예제로", "배", "배우", "배우는", "검", "검색", "검색엔", "검색엔진"]
```

edge ngram back은 edge ngram과 같은 방식으로 동작하나 색인어의 순서가 역순이 됩니다. 이 analyzer를 후방 일치에 사용하려면 edge ngram 필터 옵션 중 `side:Back`을 반드시 설정해야 하고, `edge_ngram_tokenizer`, `lowercase` 필터, `trim` 필터, `edgeNGram` 필터로 구성합니다.

[edge ngram back 색인 결과]

```
min_gram: 1
max_gram: 5
text: 실무 예제로 배우는 검색엔진
terms: ["실", "무", "실무", "예", "제", "예제", "로", "제로", "예제로", "배", "우", "배우", "는", "우는", "배우는", "검", "색", "검색", "엔", "색엔", "검색엔", "진", "엔진", "색엔진", "검색엔진"]
```

ngram과 edge ngram back의 결과를 보면 추출 방식에서 차이가 있는 것을 알 수 있습니다. 이해를 돕기 위해 '검색엔진활용'으로 ngram과 edge ngram back의 결과를 비교하면 추출된 색인어의 차이를 확인할 수 있습니다.

표 1-1 ngram과 edge ngram back 결과 비교

ngram	edge ngram back
min_gram: 1 max_gram: 5 text: 검색엔진활용 terms: [“검”, “검색”, “검색엔”, “검색엔진”, “검색엔진활”, “색”, “색엔”, “색엔진”, “색엔진활”, “색엔진활용”, “엔”, “엔진”, “엔진활”, “엔진활용”, “진”, “진활”, “진활용”, “활”, “활용”, “용”]	min_gram: 1 max_gram: 5 text: 검색엔진활용 terms: [“검”, “색”, “검색”, “엔”, “색엔”, “검색엔”, “진”, “엔진”, “색엔진”, “검색엔진”, “활”, “진활”, “엔진활”, “색엔진활”, “검색엔진활”]

1.1.2 자동 완성 예제

자동 완성 기능을 테스트하려면 analyzer 구성과 함께 term 쿼리를 이용하는 방법과 prefix 쿼리를 이용하는 방법이 있습니다.

자동 완성 설정

먼저 테스트를 위한 indice 생성에 필요한 settings와 mappings 설정을 살펴보겠습니다. 설정 관련 자세한 내용은 제공되는 소스 코드[04]를 참고하기 바라며 여기서는 일부 코드만 다루겠습니다.

[Analyzer 필드 구성 – schema/autocompletion.json]

```
"keyword" : {
    "type" : "string", "store" : "no", "index" : "analyzed", "omit_norms" : true, "index_
options" : "offsets", "term_vector" : "with_positions_offsets", "include_in_all" : false,
    "fields" : {
        "keyword_prefix" : {"type" : "string", "store" : "no", "index" : "not_analyzed",
"omit_norms" : true, "index_options" : "offsets", "term_vector" : "with_positions_offsets",
"include_in_all" : false},
        "keyword_edge" : {"index_analyzer" : "edge_ngram_analyzer", "type" : "string",
"store" : "no", "index" : "analyzed", "omit_norms" : true, "index_options" : "offsets",
"term_vector" : "with_positions_offsets", "include_in_all" : false},
        "keyword_edge_back" : {"index_analyzer" : "edge_ngram_analyzer_back", "type" :
"string", "store" : "no", "index" : "analyzed", "omit_norms" : true, "index_options" :
"offsets", "term_vector" : "with_positions_offsets", "include_in_all" : false}
    }
},
```

다양한 검색 옵션을 적용하기 위해 keyword 필드를 멀티 필드로 구성하고, 각 필드의 검색 옵션은 유형별로 구성합니다.

- **keyword_prefix :** Prefix 쿼리에서 사용하기 위한 필드
- **keyword_edge :** Term 쿼리를 이용한 전방 일치용 필드
- **keyword_edge_back :** Term 쿼리를 이용한 후방 일치용 필드

04 http://bit.ly/1w0ka01

edge ngram analyzer와 비교하기 위해 구성한 설정입니다.

[ngram_analyzer settings]

```
"ngram_analyzer" : {
    "type" : "custom",
    "tokenizer" : "ngram_tokenizer",
    "filter" : ["lowercase", "trim"]
}
```

term 쿼리를 이용하여 전방 일치를 구현하는 설정입니다.

[edge_ngram_analyzer settings]

```
"edge_ngram_analyzer" : {
    "type" : "custom",
    "tokenizer" : "edge_ngram_tokenizer",
    "filter" : ["lowercase", "trim"]
}
```

term 쿼리를 이용하여 후방 일치를 구현하는 설정입니다.

[edge_ngram_analyzer_back settings]

```
"edge_ngram_analyzer_back" : {
    "type" : "custom",
    "tokenizer" : "edge_ngram_tokenizer",
    "filter" : ["lowercase", "trim", "edge_ngram_filter_back"]
}
```

tokenizer의 타입은 `nGram`, token의 최소 길이는 1, 최대 길이는 5로 설정합니다.

[ngram_tokenizer settings]

```
"ngram_tokenizer" : {
    "type" : "nGram",
    "min_gram" : "1",
    "max_gram" : "5",
    "token_chars": [ "letter", "digit", "punctuation", "symbol" ]
}
```

ngram tokenizer와 설정은 거의 동일하며 타입만 edgeNGram으로 설정합니다.

[edge_ngram_tokenizer setting]

```
"edge_ngram_tokenizer" : {
    "type" : "edgeNGram",
    …중략…
}
```

전방 일치 기능을 구현하기 위해 side 값을 front로 설정합니다.

[edge_ngram_filter_front setting]

```
"edge_ngram_filter_front" : {
    "type" : "edgeNGram",
    "min_gram" : "1",
    "max_gram" : "5",
    "side" : "front"
}
```

후방 일치 기능을 구현하기 위해 side 값을 back으로 설정합니다.

[edge_ngram_filter_back setting]

```
"edge_ngram_filter_back" : {
    "type" : "edgeNGram",
    "min_gram" : "1",
    "max_gram" : "5",
    "side" : "back"
}
```

REST API를 이용하여 생성합니다.

[자동 완성 indice 생성]

```
$ curl -XPUT http://localhost:9200/autocompletion -d @autocompletion.json
```

전체 데이터는 소스 코드를 참고하기 바랍니다.

[자동 완성 데이터 등록 - data/autocompletion.json]

```
{ "index" : { "_index" : "autocompletion", "_type" : "search_keyword" } }
{ "keyword_id" : 1, "keyword" : "open source search engine", "keyword_ranking" : 20}
{ "index" : { "_index" : "autocompletion", "_type" : "search_keyword" } }
{ "keyword_id" : 2, "keyword" : "elasticsearch", "keyword_ranking" : 30}
…중략…
```

REST API를 이용하여 등록합니다.

```
$ curl -s -XPOST http://localhost:9200/autocompletion/_bulk --data-binary @autocompletion.
data.json
```

자동 완성 테스트 코드

여기서는 prefix 쿼리, ngram, edge ngram을 이용한 전방 일치와 edge ngram back을 이용한 후방 일치 예제를 살펴보겠습니다.

prefix 쿼리는 전방 일치에 사용할 수 있지만 일치된 색인어에 대한 강조를 적용할 수 없습니다. 이는 해당 필드에 대한 인덱스 설정을 not_analyzed로 구성하여 전체 값을 강조 처리하기 때문입니다.

[Prefix 쿼리 예제 코드]

```
Settings settings = Connector.buildSettings("elasticsearch");
Client client = Connector.buildClient(settings, new String[] {"localhost:9300"});

PrefixQueryBuilder queryBuilder = new PrefixQueryBuilder("keyword_prefix", "elastic");
String searchResult = Operators.executeQuery(settings, client, queryBuilder,
"autocompletion");
```

결과를 보면 keyword 필드에 'elastic'으로 시작하는 문서가 매칭된 것을 확인할 수 있습니다.

Prefix 쿼리 예제 결과

```
"hits" : {
```

```
    "total" : 4,
    "max_score" : 1.0,
    "hits" : [ {
        "_index" : "autocompletion",
        "_type" : "search_keyword",
        "_id" : "3",
        "_score" : 1.0,
        "_source":{ "keyword_id" : 3, "keyword" : "elasticsearch vs solr", "keyword_
ranking" : 10}
    },
    …중략…
    ]
}
```

다음 예제는 매칭된 색인어를 정확히 구분하기 위해 강조 기능을 추가하였습니다. analyzed 속성을 갖는 keyword 필드에 term 쿼리를 실행한 예제로, 추출된 색인어를 ngram 분석으로 매칭합니다. 즉, 'ucene'으로는 매칭되지만 'lucene'으로는 매칭되지 않습니다. 이는 max_gram을 5로 설정하였기 때문입니다. 여기서 검색 색인어를 'lucen' 대신 'ucene'로 한 이유는 edge ngram과 구분하기 위해서입니다.

[ngram을 이용한 질의 예제 코드]

```
Settings settings = Connector.buildSettings("elasticsearch");
Client client = Connector.buildClient(settings, new String[] {"localhost:9300"});

TermQueryBuilder queryBuilder = new TermQueryBuilder("keyword", "ucene");
String searchResult = Operators.executeQueryHighlight(settings, client, queryBuilder,
"autocompletion", "keyword", "strong");
```

highlight 영역에 strong 태그로 강조된 것을 확인할 수 있습니다.

ngram을 이용한 질의 예제 결과

```
"hits" : {
    "total" : 2,
    "max_score" : 1.4054651,
    "hits" : [ {
```

```
      "_index" : "autocompletion",
      "_type" : "search_keyword",
      "_id" : "4",
      "_score" : 1.4054651,
      "_source":{ "keyword_id" : 4, "keyword" : "lucene based search engine", "keyword_
ranking" : 10},
      "highlight" : {
        "keyword" : [ "l<strong>ucene</strong> based search engine" ]
      }
    }, {
      "_index" : "autocompletion",
      "_type" : "search_keyword",
      "_id" : "5",
      "_score" : 1.4054651,
      "_source":{ "keyword_id" : 5, "keyword" : "elasticsearch based on lucene", "keyword_
ranking" : 10},
      "highlight" : {
        "keyword" : [ "elasticsearch based on l<strong>ucene</strong>" ]
      }
    } ]
}
```

다음 예제는 ngram과 비교하기 위해 작성하였습니다. ngram에서는 'ucene'이라는 키워드로 질의하였고 edge ngram에서는 'lucen'이라는 키워드로 질의합니다. 이 둘의 차이는 앞에서 ngram_analyzer와 edge_ngram_analyzer 부분에서 설명한 내용에서 참고하기 바랍니다.

[edge ngram을 이용한 질의 예제 코드]

```
Settings settings = Connector.buildSettings("elasticsearch");
Client client = Connector.buildClient(settings, new String[] {"localhost:9300"});

TermQueryBuilder queryBuilder = new TermQueryBuilder("keyword_edge", "lucen");
String searchResult = Operators.executeQueryHighlight(settings, client, queryBuilder,
"autocompletion", "keyword_edge", "strong");
```

edge ngram을 이용한 질의 예제 결과

```
"hits" : {
```

```
    "total" : 2,
    "max_score" : 1.4054651,
    "hits" : [ {
      "_index" : "autocompletion",
      "_type" : "search_keyword",
      "_id" : "4",
      "_score" : 1.4054651,
      "_source":{ "keyword_id" : 4, "keyword" : "lucene based search engine", "keyword_
ranking" : 10},
      "highlight" : {
        "keyword_edge" : [ "<strong>lucen</strong>e based search engine" ]
      }
    }, {
      "_index" : "autocompletion",
      "_type" : "search_keyword",
      "_id" : "5",
      "_score" : 1.4054651,
      "_source":{ "keyword_id" : 5, "keyword" : "elasticsearch based on lucene", "keyword_
ranking" : 10},
      "highlight" : {
        "keyword_edge" : [ "elasticsearch based on <strong>lucen</strong>e" ]
      }
    } ]
}
```

다음은 후방 일치에 대한 예제 코드입니다. 질의어를 'e' 한 글자로 설정하였으므로 결과에서 강조된 색인어의 제일 뒤에 위치한 문자는 'e'가 됩니다.

[edge ngram을 이용한 후방 일치 예제 코드]

```
Settings settings = Connector.buildSettings("elasticsearch");
Client client = Connector.buildClient(settings, new String[] {"localhost:9300"});

TermQueryBuilder queryBuilder = new TermQueryBuilder("keyword_edge_back", "e");
String searchResult = Operators.executeQueryHighlight(settings, client, queryBuilder,
"autocompletion", "keyword_edge_back", "strong");
```

앞에서 설명한 것과 같이 강조된 글자의 마지막 문자가 'e'로 끝난 것을 확인할 수 있습니다.

edge ngram을 이용한 후방 일치 예제 결과

```
"hits" : {
    "total" : 10,
    "max_score" : 1.4246359,
    "hits" : [ {
        "_index" : "autocompletion",
        "_type" : "search_keyword",
        "_id" : "4",
        "_score" : 1.4246359,
        "_source":{ "keyword_id" : 4, "keyword" : "lucene based search engine", "keyword_
ranking" : 10},
        "highlight" : {
            "keyword_edge_back" : [ "<strong>luce</strong>ne <strong>base</strong>d
<strong>se</strong>arch <strong>e</strong>ngine" ]
        }
    },
    …중략…
    ]
}
```

한글 후방 일치는 네이버 메인 검색창 또는 지식쇼핑 검색창에 '청바지'라는 검색어를 넣으면 간단히 확인할 수 있습니다. 다음 그림의 블록 지정된 부분이 후방 일치 부분입니다.

그림 1-4 한글 후방 일치 예

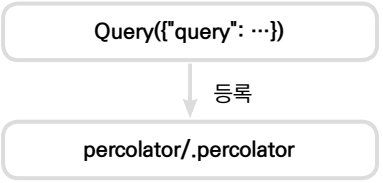
1.2 Percolator

Percolator는 문서 색인보다는 문서 모니터링에 주로 사용하며 역검색(Reverse Search)이라고도 합니다. 이 기능은 증권, 경매, 광고 등의 다양한 서비스에서 활용할 수 있고, 특정 로그에 대한 감시용으로도 사용할 수 있습니다.

Elasticsearch에서 percolator는 인덱스에 쿼리를 하나의 타입Type[05]으로 지정하여 저장합니다. 즉, percolator라는 인덱스에 `.percolator`라는 타입으로 쿼리를 저장합니다.

도큐먼트Document[06]를 색인할 때 먼저 percolator 요청을 수행하고 결과에 따른 처리를 하는데, 이 percolator로 요청하는 과정이 역검색입니다. 다음 그림은 percolator의 논리적인 개념을 보여줍니다.

그림 1-5 Percolator 생성과 쿼리 등록



구현 방법은 요구 사항에 따라 달라집니다. 첫 번째 방법은 [그림 1-6]처럼 발생한 문서가 등록한 percolator 쿼리에 매치되는지 질의한 후 결과에 따라 Alert 처리를 하거나 색인Indexing을 수행하도록 구현합니다. 목적에 따라서는 둘 다 수행할 수도 있습니다. 두 번째 방법은 [그림 1-7]처럼 발생한 문서를 먼저 색인한 후 percolator 쿼리에 매치되는지 질의해서 결과에 따라 Alert를 수행합니다.

05 도큐먼트 타입은 물리적인 인덱스나 저장소를 가지고 있지 않다. 다만 논리적으로 단일 인덱스에 대한 서로 다른 목적의 데이터를 구분하여 저장하는 방법으로 사용된다. 데이터베이스 관점에서 보면 테이블과 유사하며, 내장 필드인 _type에 따라 저장된다.

06 검색에서 가장 기본이 되는 데이터 단위로, Elasticsearch에 저장되는 하나의 아이템 또는 아티클(article)을 말한다. 도큐먼트는 RDBMS에서 테이블 내 하나의 row에 해당한다.

그림 1-6 Percolator 구현 방법 1

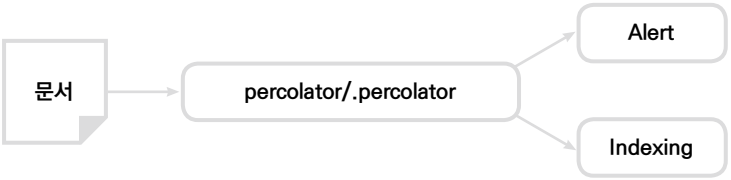
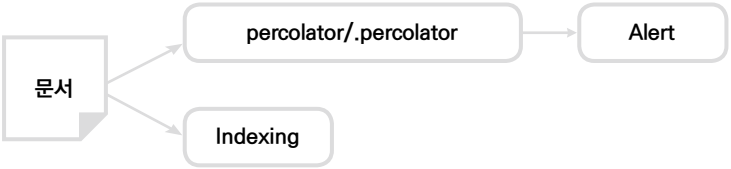


그림 1-7 Percolator 구현 방법 2

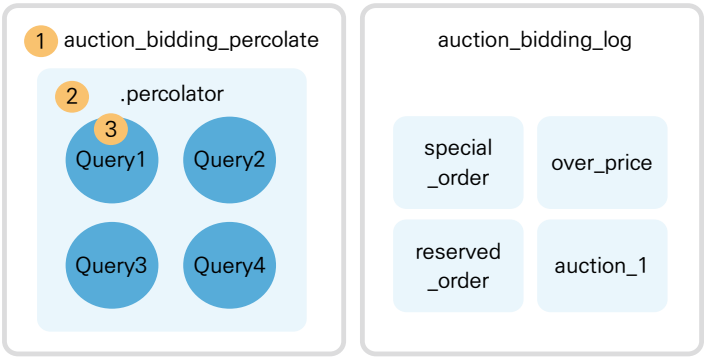


1.2.1 Percolator 생성

이번 예제는 온라인 경매 서비스에서 입찰 금액을 실시간으로 모니터링하기 위한 구성으로 작성되었습니다. Percolator 쿼리를 등록하기 위한 인덱스는 auction_bidding_percolate로 생성하고 percolator 쿼리에 매칭된 도큐먼트는 auction_bidding_log로 생성된 인덱스에 등록합니다.

percolator 쿼리는 총 4개고, 각 쿼리에 매칭된 도큐먼트는 auction_bidding_log의 타입에 맞춰서 유형별로 등록합니다.

그림 1-8 Percolator index



① 번은 인덱스를 의미합니다.

② 번은 타입을 의미합니다. Percolator 타입은 `.percolator`로 생성됩니다.

③ 번은 도큐먼트를 의미합니다. Percolator는 쿼리 자체가 하나의 도큐먼트가 됩니다.

다음 두 예제는 경매 서비스에서 특정 키워드와 범위(range) 조건을 지정하고 이 조건과 일치할 때 모니터링을 수행합니다.

이 쿼리는 경매 입찰 시 'special'과 'order'라는 두 개의 키워드가 포함되어 있을 때 해당 입찰 건을 모니터링합니다. inner query key 영역("query" : "special order")의 값을 'over price'와 'reserved order'로 변경하여 키워드 모니터링용 percolator를 생성할 수 있게 합니다.

[예제 1. 키워드 모니터링]

```
{
    "query" : {
        "match" : {
            "bidding_keyword" : {
                "query" : "special order",
                "operator" : "and"
            }
        }
    }
}
```

이 쿼리는 범위(range) 모니터링 예제로 경매에 1번 참여할 때 입찰 금액이 1천만 원을 초과한 입찰 건을 모니터링합니다.

[예제 2. 범위 모니터링]

```
{
    "query": {
        "bool": {
            "must": [
                {
                    "term": {
                        "auction_id": 1
```

```
                    }
                },
                {
                    "range": {
                        "bidding_price": {
                            "gt": 10000000
                        }
                    }
                }
            ]
        }
    }
}
```

auction_bidding_log 인덱스에 생성한 타입들은 특정 조건과 일치하는 경매 입찰이 들어왔을 때 해당 이벤트를 유형별로 기록합니다.

[Percolator 인덱스와 타입 생성 예제 코드]

```
Settings settings = Connector.buildSettings("elasticsearch");
Client client = Connector.buildClient(settings, new String[] {"localhost:9300"});
String setting = "";
String[] mapping = new String[4];

setting = Operators.readFile("schema/percolate_settings.json");
mapping[0] = Operators.readFile("schema/percolate_mappings_special_order.json");
mapping[1] = Operators.readFile("schema/percolate_mappings_over_price.json");
mapping[2] = Operators.readFile("schema/percolate_mappings_reserved_order.json");
mapping[3] = Operators.readFile("schema/percolate_mappings_auction_1.json");

try {
    client.admin().indices().delete(new DeleteIndexRequest("auction_bidding_log")).
actionGet();
    client.admin().indices().delete(new DeleteIndexRequest("auction_bidding_percolate")).
actionGet();
} catch (Exception e) {
} finally {
}

CreateIndexResponse createIndexResponse = client.admin().indices()
    .prepareCreate("auction_bidding_log")
    .setSettings(setting)
    .addMapping("special_order", mapping[0])
```

```
    .addMapping("over_price", mapping[1])
    .addMapping("reserved_order", mapping[2])
    .addMapping("auction_1", mapping[3])
    .execute()
    .actionGet();

createIndexResponse = client.admin().indices()
    .prepareCreate("auction_bidding_percolate")
    .setSettings(setting)
    .execute()
    .actionGet();

client.close();
```

이벤트 유형에 따라 등록되는 타입은 다음 표와 같습니다.

표 1-2 유형별 등록 타입

유형	타입
special order	special_order
over price	over_price
reserved order	reserved_order
경매 1번의 입찰 가격이 1천만 원을 초과할 때	auction_1

1.2.2 Percolator Query 등록

색인 문서를 필터링 또는 모니터링하려면 검색 쿼리를 등록해야 합니다. 여기서는 경매 입찰 건의 모니터링 조건을 `auction_bidding_percolate` 인덱스의 `.percolator` 타입에 등록합니다. Percolator 쿼리는 XContentBuilder와 JSON string의 두 가지 방법으로 등록할 수 있는데, XContentBuilder는 setSource(source), JSON string은 setSource(json)으로 등록합니다.

Percolator 요청(request) 조건이 일치할 때 `auction_bidding_log`의 타입과 일치시키기 위해 percolator 쿼리 등록 시 도큐먼트 id에 `auction_bidding_log` 타입을 등록합니다.

다음 코드는 경매 입찰 시 등록한 키워드에 'special'과 'order'라는 키워드가 모두 포함되어 있으면 해당 입찰에 대해 정의한 액션을 수행합니다. 'special order'는 'over price'와 'reserved order'로 수정하여 등록할 수 있게 합니다.

[예제 1. 등록 코드]

```
// XContentBuilder 이용.
QueryBuilder queryBuilder = QueryBuilders.matchQuery("bidding_keyword", "special order").
operator(Operator.AND);
XContentBuilder json = jsonBuilder().startObject();
    json.field("query", queryBuilder);
json.endObject();

// JSON string 이용
MatchQueryBuilder matchQueryBuilder = new MatchQueryBuilder("bidding_keyword", "special
order");
matchQueryBuilder.operator(Operator.AND);
String source = "{\"query\" : " + matchQueryBuilder.toString() + "}";

client.prepareIndex("auction_bidding_percolate", ".percolator", "special_order")
//   .setSource(source)
    .setSource(json)
    .execute()
    .actionGet();
```

다음 코드는 범위에 해당하는 값을 모니터링하여 경매 1번에 입찰한 금액이 1천만 원을 초과할 때 정의된 액션을 수행합니다. 예제 코드는 제공된 소스 코드를 참고하여 생성하기 바랍니다.

[예제 2. 등록 코드]

```
TermQueryBuilder termQueryBuilder = new TermQueryBuilder("auction_id", 1);
RangeQueryBuilder rangeQueryBuilder = new RangeQueryBuilder("bidding_price");
rangeQueryBuilder.gt(10000000);
queryBuilder = QueryBuilders.boolQuery().must(rangeQueryBuilder).must(termQueryBuilder);

json = jsonBuilder().startObject();
    json.field("query", queryBuilder);
json.endObject();

client.prepareIndex("auction_bidding_percolate", ".percolator", "auction_1")
    .setSource(json)
```

```
    .execute()
    .actionGet();
```

1.2.3 Percolator 요청

Percolate 인덱스를 생성하고 쿼리 등록까지 끝났으니 이제 percolator 요청를 생성하여 요청이 어떻게 동작하는지 알아보겠습니다. 먼저 경매 입찰 요청이 입력되면 등록된 percolator 쿼리로 이 요청을 보내 일치 여부에 따른 액션을 수행합니다.

다음은 발생한 이벤트에 대한 percolator 요청을 수행한 후 결과를 획득하는 예로, 'special order'와 'auction_1' 두 가지 경우를 포함하고 있습니다. 이 예제의 결과값에 따른 처리는 [응답 후 처리 코드]를 참고하기 바랍니다.

[요청 코드]

```
PercolateRequestBuilder precolateRequestBuilder = new PercolateRequestBuilder(client);
// 가상의 경매 입찰 문서를 생성합니다.
DocBuilder docBuilder = new DocBuilder();
XContentBuilder jsonDoc = jsonBuilder().startObject()
        .field("auction_id", 1)
        .field("bidding_id", 1)
        .field("bidding_keyword",  "special order")
        .field("bidding_price", 200000000)
    .endObject();

docBuilder.setDoc(jsonDoc);

// percolator request를 보낸다.
PercolateResponse percolateResponse = precolateRequestBuilder.setIndices("auction_
bidding_percolate")
        .setDocumentType(".percolator")
        .setPercolateDoc(docBuilder)
        .execute()
        .actionGet();
```

다음은 두 가지 조건에 대한 percolator 요청을 수행한 후 결과에 따른 색인 방

법을 보여주기 위한 예로, 색인 결과는 [요청 코드]의 'special order' 조건에 따라 special_order 타입에, auction_id:1, bidding_price:200000000 조건에 따라 auction_1 타입에 등록됩니다.

[응답 후 처리 코드]

```
Match[] matches = percolateResponse.getMatches();
int size = matches.length;

for ( int i=0; i<size; i++ ) {
    String docType = matches[i].getId().string();

    if ( "auction_1".equalsIgnoreCase(docType) ) {
        IndexRequestBuilder requestBuilder;
        IndexResponse response;

        requestBuilder = client.prepareIndex("auction_bidding_log", docType);
        response = requestBuilder
                .setSource(jsonDoc)
                .execute()
                .actionGet();
    }

    if ( "special_order".equalsIgnoreCase(docType) ) {
        IndexRequestBuilder requestBuilder;
        IndexResponse response;

        requestBuilder = client.prepareIndex("auction_bidding_log", docType);
        response = requestBuilder
                .setSource(jsonDoc)
                .execute()
                .actionGet();
    }
}
```

키워드 모니터링을 하려면 'reserved order'와 'over price'는 'special order'와 같은 방법으로 타입을 매칭시켜 처리하면 됩니다.

1.3 Join

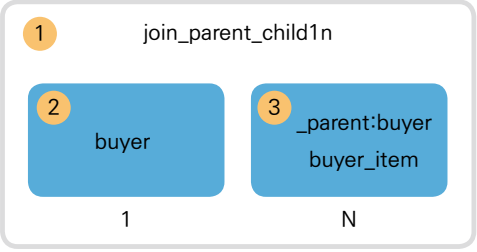
Elasticsearch는 정규화된 데이터Normalized data를 다루는 RDBMS와 다르게 비정규화된 데이터Denormalized data로 문서가 구성되어서 RDBMS와 같은 조인 기능을 제공하기가 쉽지 않습니다. 여기서는 조인 기능을 Elasticsearch에서 어떻게 처리하는지 알아보겠습니다.

Elasticsearch에서는 크게 두 가지 방법으로 조인 기능을 구현합니다.

애플리케이션 단에서의 조인(Application-side Joins)

이 방법은 두 개의 테이블에서 외래키Foreign Key 정보를 N 관계에 있는 테이블에 함께 저장하여 관계 모델을 구성합니다. Elasticsearch에서는 parent-child 관계를 이용하는데, 한 인덱스에 parent 타입과 child 타입을 등록하여 관계 모델을 구현합니다. 예를 들어, 1의 관계를 정의하기 위한 parent 타입으로 `buyer`를 생성하고 구매자 정보를 저장하며, N의 관계를 정의하기 위한 child 타입으로 `buyer_item`을 생성하고 구매자의 구매 이력을 저장합니다.

그림 1-9 parent-child 관계 모델



① 번은 1:N 관계를 포함하는 인덱스입니다.

② 번은 parent 타입으로 1의 관계를 가지며, child 타입의 외래키 값을 포함합니다.

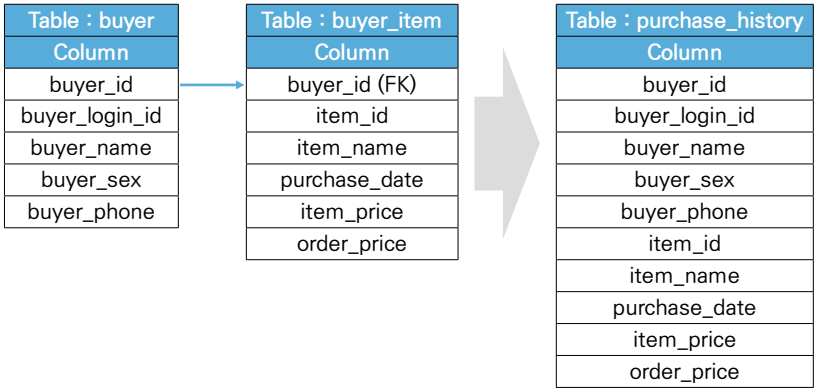
③ 번은 child 타입으로 N의 관계를 가지며, _parent.type 필드에 parent 타입을 지정해야 합니다.

비정규화 데이터(Denormalizing Data)

이 방법은 관계 데이터를 중복으로 구성하여 하나의 테이블에 모든 데이터를 등록

하는 것인데, 색인 크기가 증가한다는 점을 유의해야 합니다. Elasticsearch에서는 비정규화 데이터를 구성하기 위해 inner objects와 nested 타입을 제공하므로 인덱스의 스키마를 구성하기 위한 매핑 정보 설정 시 관계 데이터에 대한 필드를 object 타입 또는 nested 타입으로 정의하여 색인 시 모든 데이터를 등록합니다.

그림 1-10 비정규화 모델



NOTE Field Collapsing

앞의 두 가지 방법 외에 추가로 두 방법을 기반으로 aggregation을 적용하여 조인 기능을 구현하는 방법이 있습니다. child 타입에 aggregation하기 위한 최소 정보를 parent 타입에서 가져와 색인 시 함께 등록하고, 질의 시 aggregation 질의를 통해 결과를 가져오는 방법인데, 일반적인 화면 구성을 위한 결과로 사용할 수 없으므로 여기서는 간단히 소개만 했습니다.

1.3.1 Parent-Child

parent-child 타입은 반드시 같은 인덱스에 생성해야 하고, 서로 다른 인덱스 생성해서 사용할 수 없습니다. 또한, 인덱스 단위로 선언하는 것도 불가능합니다.

parent 타입은 반드시 기본키primary key 역할을 하는 _id 값을 지정해야 하는데, child 타입에서 문서를 등록할 때 이 값을 _parent 필드의 외래키로 반드시 설

정해야 합니다. child 타입을 정의할 때 매핑 설정에서 `_parent.type` 값은 앞에서 생성한 parent 타입명으로 지정해야만 정상적으로 parent-child 타입을 사용할 수 있습니다.

parent type : buyer

- 구매자의 기본 정보 또는 메타 데이터를 저장하며, 구매자 한 명의 정보는 고유합니다.

child type : buyer_item

- 구매상품에 구매정보를 저장하며, 구매자별로 복수 개의 구매정보가 있습니다.

Elasticsearch에서는 parent-child 기능 구현을 위해 다음 두 종류의 API를 제공합니다.

has_parent 쿼리/필터

- 이 API는 parent 도큐먼트에 질의하고 child 도큐먼트를 반환합니다.

has_child 쿼리/필터

- 이 API는 child 도큐먼트에 질의하고 parent 도큐먼트를 반환합니다.

다음 예제로 parent-child 타입 생성과 구성을 확인해 보겠습니다(전체 코드는 제공된 소스 코드[07]를 참고하기 바랍니다).

[Parent-Child 인덱스 생성 - settings]

```
"settings" : {
    "number_of_shards" : 3,
    "number_of_replicas" : 0,
    "index" : {
        …중략…
    }
}
```

07 http://bit.ly/1zHvFob