



Hanbit eBook

Realtime 93

실무 예제로 배우는

Elasticsearch 검색엔진

활용편

정호욱 지음

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** 김태현

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 최승실

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

Copyright © 2015 정호욱 & HANBIT Media, Inc.

이 책의 저작권은 정호욱과 한빛미디어(주)에 있습니다.

저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

지금 하지 않으면 할 수 없는 일이 있습니다.

책으로 떠내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.

한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

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 그루터^{Gruter}에서 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. 루센^{Lucene}이라는 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



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 이해하는데 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.

무료로 업데이트되는 전자책 전용 서비스입니다

2

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.

지금 보시는 전자책에 소유권한을 표시한 문구가 없거나 타인의 소유권한을 표시한 문구가 있다면 위법하게 사용하고 있을 가능성이 큽니다. 이 경우 저작권법에 따라 불이익을 받으실 수 있습니다.

다양한 기기에 사용할 수 있습니다. 또한, 한빛미디어 사이트에서 구매하신 후에는 횟수에 관계없이 내려받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 불이기가 가능합니다.

전자책은 오픈자 교정이나 내용의 수정·보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려 드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구매하신 독자에게만 제공되며, 다른 사람에게 양도나 이전은 허락되지 않습니다.

chapter 1 검색 기능 확장 —— 001

1.1 자동 완성 ———	001
1.1.1 자동 완성 Analyzer ———	002
1.1.2 자동 완성 예제 ———	005
1.2 Percolator ———	013
1.2.1 Percolator 생성 ———	014
1.2.2 Percolator Query 등록 ———	017
1.2.3 Percolator 요청 ———	019
1.3 Join ———	021
1.3.1 Parent-Child ———	022
1.3.2 Nested ———	029
1.4 River ———	035
1.4.1 JDBC River 동작의 이해 ———	035
1.4.2 JDBC River 설치 전 준비작업 ———	037
1.4.3 JDBC River 설치 ———	039
1.4.4 JDBC River indice 구성 ———	040
1.4.5 JDBC River 등록 ———	041
1.4.6 JDBC River 실행 ———	047
1.4.7 JDBC River에서 REST API 이용하기 ———	048
1.5 정리 ———	049

2.1 Bucket Aggregation	052
2.1.1 Global Aggregation	053
2.1.2 Filter Aggregation	055
2.1.3 Missing Aggregation	056
2.1.4 Nested Aggregation	057
2.1.5 Reverse Nested Aggregation	058
2.1.6 Terms Aggregation	060
2.1.7 Significant Terms Aggregation	063
2.1.8 Range Aggregation	065
2.1.9 Date Range Aggregation	067
2.1.10 Histogram Aggregation	069
2.1.11 Date Histogram Aggregation	071
2.1.12 Geo Distance Aggregation	072
2.2 Metric Aggregation	074
2.2.1 Min Aggregation	074
2.2.2 Max Aggregation	075
2.2.3 Sum Aggregation	076
2.2.4 Avg Aggregation	077
2.2.5 Stats Aggregation	078
2.2.6 Extended Stats Aggregation	079
2.2.7 Value Count Aggregation	080
2.2.8 Percentiles Aggregation	081
2.2.9 Cardinality Aggregation	082
2.3 정리	083

chapter 3 Plugin —— 085

3.1 Plugin 제작 —————	085
3.1.1 Plugin 프로젝트 생성 —————	085
3.1.2 Plugin 프로젝트 구성 —————	086
3.2 REST Plugin 만들기 —————	088
3.2.1 REST Plugin 프로젝트 생성과 등록 —————	088
3.2.2 REST Plugin 기능 구현 —————	090
3.2.3 REST Plugin 등록 설정 —————	090
3.2.4 REST Plugin 빌드와 설치 —————	091
3.2.5 REST Plugin 구현 요약 —————	093
3.3 Analyzer Plugin 만들기 —————	094
3.3.1 Analyzer Plugin 프로젝트 생성과 등록 —————	094
3.3.2 Analyzer Plugin 기능 구현 —————	095
3.3.3 Analyzer Plugin 등록 설정 —————	098
3.3.4. Analyzer Plugin 빌드와 설치 —————	100
3.4 정리 —————	103

chapter 4 Hadoop 연동 —— 105

4.1 MapReduce 연동 —————	107
4.1.1 준비 항목 —————	108
4.1.2 색인 MapReduce 구현 —————	108
4.1.3 검색 MapReduce 구현 —————	111
4.2 Hive 연동 —————	114
4.2.1 준비 항목 —————	114
4.2.2 색인 구현 —————	115
4.2.3 검색 구현 —————	118
4.3 정리 —————	120



chapter 5 ELK 연동 ————— 121

5.1 Logstash	121
5.1.1 다운로드와 설치	122
5.1.2 실행과 테스트	123
5.1.3 Command-line Flag 알아보기	124
5.1.4 Config 알아보기	126
5.1.5 Input 알아보기	129
5.1.6 Filter 알아보기	130
5.1.7 Output 알아보기	132
5.1.8 Codec 알아보기	134
5.2 Elasticsearch	135
5.2.1 다운로드와 설치	135
5.2.2 실행과 테스트	136
5.2.3 기본 플러그인 설치	136
5.3 Kibana	137
5.3.1 다운로드와 설치	137
5.3.2 실행	138
5.3.3 대시보드 만들기	140
5.4 정리	148

chapter 6 SQL 활용하기 ————— 149

6.1 RDB 관점의 Elasticsearch	150
6.1.1 Index vs. Database	150
6.1.2 Type vs. Table	150
6.1.3 Document vs. Row	151
6.1.4 Field vs. Column	151
6.1.5 Analyzer vs. Index	151
6.1.6 _id vs Primary Key	151

6.1.7 Mapping vs. Schema	152
6.1.8 Shard/Route vs. Partition	152
6.1.9 Parent–Child/Nested vs. Relation	152
6.1.10 Query DSL vs. SQL	153
6.2 SQL 정의하기	153
6.2.1 SQL 정의	153
6.2.2 인덱스 생성/삭제/선택	157
6.2.3 테이블 생성/삭제	158
6.2.4 절 선택/삽입/업데이트/삭제	159
6.3 SQL 변환하기	160
6.3.1 Match_all Query	160
6.3.2 Match Query	161
6.3.3 Bool Query	161
6.3.4 Ids Query	165
6.3.5 Range Query	165
6.3.6 Term Query	167
6.3.7 Terms Query	167
6.4 JDBC Driver 만들기	168
6.4.1 Elasticsearch Driver	169
6.4.2 Elasticsearch Connection	169
6.4.3 Elasticsearch Statement	170
6.4.4 Elasticsearch ResultSet	171
6.4.5 Elasticsearch ResultSetMetaData	172
6.4.6 Elasticsearch JDBC Driver 예제	173
6.5 정리	174



chapter 7 Elasticsearch 성능 최적화 —— 175

7.1 하드웨어 관점 —————	175
7.1.1 CPU —————	176
7.1.2 RAM —————	178
7.1.3 DISK —————	179
7.1.4 NETWORK —————	179
7.2 Document 관점 —————	180
7.2.1 Index와 Shard 튜닝 —————	180
7.2.2 Modeling —————	182
7.3 Operation 관점 —————	186
7.3.1 설정 튜닝 —————	186
7.3.2 검색 튜닝 —————	192
7.3.3 색인 튜닝 —————	195
7.4 정리 —————	197

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.

검색 기능 확장

1.1 자동 완성

자동 완성^{Auto 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 토큰^{Token} 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 필드^{Field}⁰¹의 인덱스^{Index}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 질의이고, 뒤 항목은 분석되어 저장된 단어가 매칭된 모습입니다.

그림 1-1 전방 일치

가					가	나	다	라
---	--	--	--	--	---	---	---	---

그림 1-2 부분 일치

나					가	나	다	라
---	--	--	--	--	---	---	---	---

그림 1-3 후방 일치

라					가	나	다	라
---	--	--	--	--	---	---	---	---

1.1.1 자동 완성 Analyzer

자동 완성용 indice⁰³를 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 크기 까지 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:Back을 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 돋기 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	min_gram: 1
max_gram: 5	max_gram: 5
text: 검색엔진활용	text: 검색엔진활용
terms: ["검", "검색", "검색엔", "검색엔진", "검색엔진활", "색", "색엔", "색엔진", "색엔진활", "색엔진활용", "엔", "엔진", "엔진활", "엔진활용", "진", "진활", "진활용", "활", "활용"]	terms: ["검", "색", "검색", "엔", "색엔", "검색엔", "엔", "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](#)⁰⁴를 참고하기 바라며 여기서는 일부 코드만 다루겠습니다.

[Analyzer 필드 구성 – schema/autocomplete.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/autocomplete -d @autocomplete.json
```

전체 데이터는 소스 코드를 참고하기 바랍니다.

[자동 완성 데이터 등록 – data/autocomplete.json]

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

REST API를 이용하여 등록합니다.

```
$ curl -s -XPOST http://localhost:9200/autocomplete/_bulk --data-binary @autocomplete.
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,
"autocomplete");
```

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" : "autocomplete",  
    "_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,  
"autocomplete", "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" : "autocomplete",
    "_type" : "search_keyword",
    "_id" : "4",
    "_score" : 1.4054651,
    "_source":{ "keyword_id" : 4, "keyword" : "lucene based search engine", "keyword_ranking" : 10},
    "highlight" : {
      "keyword" : [ "l<strong>lucene</strong> based search engine" ]
    }
  },
  {
    "_index" : "autocomplete",
    "_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>lucene</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,
"autocomplete", "keyword_edge", "strong");

```

edge ngram을 이용한 질의 예제 결과

```
"hits" : {
```

```
"total" : 2,
"max_score" : 1.4054651,
"hits" : [ {
    "_index" : "autocomplete",
    "_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" : "autocomplete",
    "_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,
"autocomplete", "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" : "autocomplete",  
        "_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

청바지

남자 청바지
여자 청바지

청바지 브랜드

데님 청바지

청바지 쇼핑몰

청바지 에어울리는신발

청바지 리폼

도움말 | 신고 검색어저장 끄기 | 자동완성 끄기

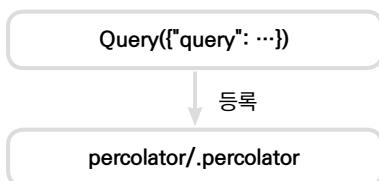
1.2 Percolator

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 타입^{Type05}으로 지정하여 저장합니다. 즉, percolator라는 인덱스에 .percolator라는 타입으로 쿼리를 저장합니다.

도큐먼트^{Document06}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 색인^{Indexing}을 수행하도록 구현합니다. 목적에 따라서는 둘 다 수행 할 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

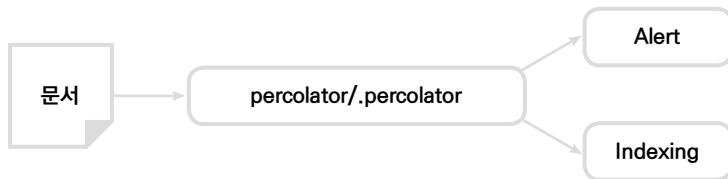
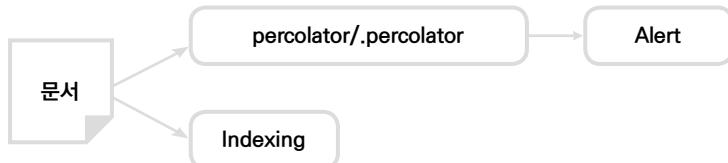


그림 1-7 Percolator 구현 방법 2

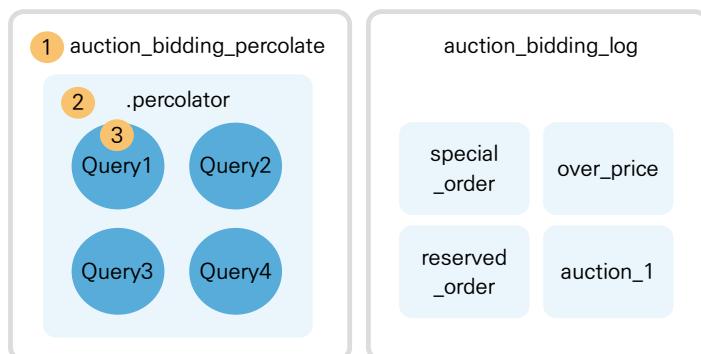


1.2.1 Percolator 생성

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 .percolator로 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 모니터링 용 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_log의 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 요청을 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

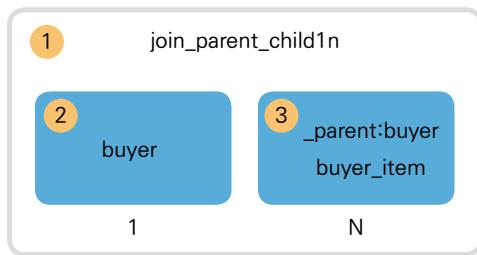
Elasticsearch는 정규화된 데이터^{Normalized data}를 다루는 RDBMS와 다르게 비정규화된 데이터^{Denormalized data}로 문서가 구성되어서 RDBMS와 같은 조인 기능을 제공하기가 쉽지 않습니다. 여기서는 조인 기능을 Elasticsearch에서 어떻게 처리하는지 알아보겠습니다.

Elasticsearch에서는 크게 두 가지 방법으로 조인 기능을 구현합니다.

애플리케이션 단에서의 조인(Application-side Joins)

이 방법은 두 개의 테이블에서 외래키^{Foreign 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 buyer를 생성하고 구매자 정보를 저장하며, N의 관계를 정의하기 위한 child 타입으로 buyer_item을 생성하고 구매자의 구매 이력을 저장합니다.

그림 1-9 parent-child 관계 모델



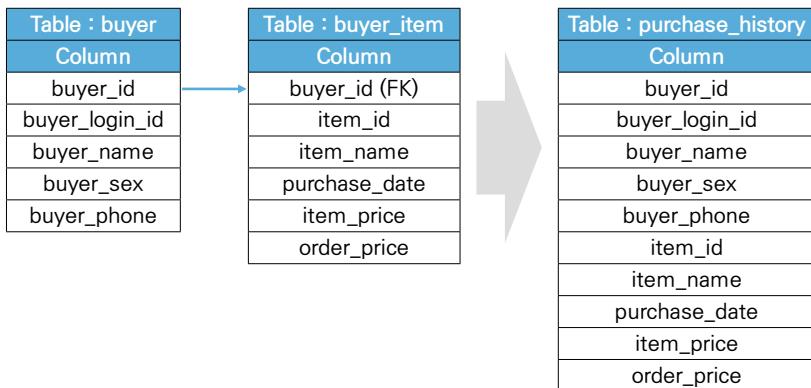
- ① 번은 1:N 관계를 포함하는 인덱스입니다.
- ② 번은 parent 타입으로 1의 관계를 가지며, child 타입의 외래키 값을 포함합니다.
- ③ 번은 child 타입으로 N의 관계를 가지며, _parent.type 필드에 parent 타입을 지정해야 합니다.

비정규화 데이터(Denormalizing Data)

이 방법은 관계 데이터를 중복으로 구성하여 하나의 테이블에 모든 데이터를 등록

하는 것인데, 색인 크기가 증가한다는 점을 유의해야 합니다. Elasticsearch에 서는 비정규화 데이터를 구성하기 위해 inner objects와 nested 타입을 제공하므로 인덱스의 스키마를 구성하기 위한 매팅 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 구성은 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 기본키 primary 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 코드⁰⁷](#)를 참고하기 바랍니다).

[Parent-Child 인덱스 생성 – settings]

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

⁰⁷ <http://bit.ly/1zHvFob>

앞에서 설명한 것처럼 기본키에 해당하는 _id 값을 설정합니다.

[Parent 탑입 생성 - mappings]

```
"buyer" : {  
    "_id" : {  
        "index" : "not_analyzed",  
        "path" : "buyer_id"  
    },  
    ...중략...  
    "properties" : {  
        "buyer_id" : {"type" : "long", "store" : "no", "index" : "not_analyzed", "index_options" : "docs", "ignore_malformed" : true, "include_in_all" : false},  
        ...중략...  
    }  
}
```

child 탑입은 parent 탑입을 반드시 지정해야 합니다. 지정하지 않아도 예러가 발생하지는 않으나 has_parent query/filter와 has_child query/filter를 사용할 수 없습니다. buy_date 필드는 화면 표시와 검색 질의 시 분리해서 사용하기 위해 멀티 필드로 구성합니다.

[Child 탑입 생성 - mappings]

```
"buyer_item" : {  
    "_parent" : {  
        "type" : "buyer"  
    },  
    ...중략...  
    "properties" : {  
        ...중략...  
        "buy_date" : {  
            "type" : "date", "format" : "yyyyMMddHHmmss", "store" : "no", "index" : "not_analyzed", "index_options" : "docs", "ignore_malformed" : true, "include_in_all" : false,  
            "fields" : {  
                "buy_date_search" : {"type" : "long", "store" : "no", "index" : "not_analyzed", "index_options" : "docs", "ignore_malformed" : true, "include_in_all" : false}  
            }  
        },  
        ...중략...  
    }  
}
```

```
    }  
}
```

parent-child 인덱스를 생성할 때 같은 인덱스에 parent와 child 타입이 생성되는 것을 확인할 수 있습니다.

[Parent-Child 인덱스 생성 코드 – JoinParentChildTest.java]

```
setting = Operators.readFile("schema/join_parent_child_settings.json");  
mapping[0] = Operators.readFile("schema/join_parent_child_mappings_parent1.json");  
mapping[1] = Operators.readFile("schema/join_parent_child_mappings_childN.json");  
  
try {  
    client.admin().indices().delete(new DeleteIndexRequest("join_parent_child1n")).  
    actionGet();  
} catch (Exception e) {  
} finally {  
}  
  
CreateIndexResponse createIndexResponse = client.admin().indices()  
    .prepareCreate("join_parent_child1n")  
    .setSettings(setting)  
    .addMapping("buyer", mapping[0])  
    .addMapping("buyer_item", mapping[1])  
    .execute()  
    .actionGet();
```

Parent-Child 도큐먼트 등록 예제

_index는 문서를 등록할 인덱스 지정하고, _type는 문서를 등록할 인덱스 내 타입을 지정합니다.

[Parent 도큐먼트 – join_parent_child1N.json]

```
{ "index" : { "_index" : "join_parent_child1n", "_type" : "buyer" } }  
{ "buyer_id" : 1, "buyer_login_id" : "jjeong", "buyer_name" : "헨리", "buyer_sex" : "M", "buyer_phone" : "01019931974" }
```

parent 타입과 달리 `_parent` 필드를 설정해야 합니다. `_parent`에는 parent 타입의 1:N 관계를 생성할 수 있는 문서의 `_id` 값을 설정해야 하는데, 여기서는 parent 타입의 `buyer_id` 값이 이에 해당합니다.

[Child 도큐먼트 – `join_parent_child1N.json`]

```
{ "index" : { "_index" : "join_parent_child1n", "_type" : "buyer_item", "_parent" : 1 } }
{"buyer_item_id" : 1,"item_id" : 1,"item_name" : "스캇 자전거","buy_date" :
"20140611121212","item_price" : 750000,"order_price" : 600000}
```

REST API를 통해 등록합니다.

[Bulk 요청]

```
curl -s -XPOST 'http://localhost:9200/join_parent_child1n/_bulk' --data-binary @join_
parent_child1N.json
```

has_parent 쿼리/필터 예제

has_parent 쿼리는 has_parent 필드의 내부에 parent_type을 지정하는데, 내부적으로 has_parent 필터와 같습니다. has_parent 쿼리는 parent 타입에 질의를 수행하며 parent 타입의 도큐먼트 `_id` 값을 가지고 child 타입의 도큐먼트를 반환합니다.

[has_parent의 Query DSL 코드 – `JoinParentChildTest.java`]

```
"query": {
    "has_parent": {
        "parent_type": "buyer",
        "query": {
            "term" : {
                "buyer_login_id" : "atie"
            }
        }
    }
}
```

앞에서 정의한 parent-child 모델에서와 같이 parent 타입의 buyer_login_id 필드에 질의하는 것을 확인할 수 있습니다.

[has_parent의 Java API 코드 – JoinParentChildTest.java]

```
String parentType = "buyer";
TermQueryBuilder termQueryBuilder = new TermQueryBuilder("buyer_login_id", "atie");
HasParentQueryBuilder hasParentQueryBuilder = new HasParentQueryBuilder(parentType,
termQueryBuilder);
String searchResult = Operators.executeQuery(settings, client, hasParentQueryBuilder,
"join_parent_child1n");
```

실행하면 다음과 같이 child 타입의 도큐먼트 결과만 나옵니다.

has_parent 쿼리/필터 실행 결과

```
"hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
        "_index" : "join_parent_child1n",
        "_type" : "buyer_item",
        "_id" : "3",
        "_score" : 1.0,
        "_source": {"buyer_item_id" : 3,"item_id" : 2,"item_name" : "자전거 트레일러","buy_date" : "20140611121212","item_price" : 3400000,"order_price" : 1950000}
    }, {
        "_index" : "join_parent_child1n",
        "_type" : "buyer_item",
        "_id" : "4",
        "_score" : 1.0,
        "_source": {"buyer_item_id" : 4,"item_id" : 3,"item_name" : "비싼 자전거","buy_date" : "20130610121212","item_price" : 75000000,"order_price" : 10000000}
    } ]
}
```

has_child 쿼리/필터 예제

has_child 쿼리는 has_child 필드와 내부에 type을 지정하는데, 내부적으로

has_child 필터와 같습니다. has_child 쿼리는 child 타입에 질의를 수행하며, child 타입의 도큐먼트에 _parent로 지정한 값에 해당하는 parent 타입의 도큐먼트를 반환합니다.

[has_child의 Query DSL 코드 – JoinParentChildTest.java]

```
"query": {  
    "has_child": {  
        "type": "buyer_item",  
        "query": {  
            "term" : {  
                "item_id" : 1  
            }  
        }  
    }  
}
```

child 타입의 buyer_item으로 질의하는 것을 확인할 수 있습니다.

[has_child의 Java API 코드 – JoinParentChildTest.java]

```
String childType = "buyer_item";  
TermQueryBuilder termQueryBuilder = new TermQueryBuilder("item_id", 1);  
HasChildQueryBuilder hasChildQueryBuilder = new HasChildQueryBuilder(childType,  
termQueryBuilder);  
String searchResult = Operators.executeQuery(settings, client, hasChildQueryBuilder,  
"join_parent_child1n");
```

실행하면 다음과 같이 parent 타입의 도큐먼트 결과만 나옵니다.

has_child 쿼리/필터 실행 결과

```
"hits" : {  
    "total" : 1,  
    "max_score" : 1.0,  
    "hits" : [ {  
        "_index" : "join_parent_child1n",  
        "_type" : "buyer",  
        "_id" : "1",  
        "_score" : 1.0,
```

```
        "_source": {"buyer_id" : 1, "buyer_login_id" : "jjeong", "buyer_name" : "헨리", "buyer_sex" : "M", "buyer_phone" : "01019931974"}  
    } ]  
}
```

1.3.2 Nested

정규화된 데이터를 비정규화 과정을 통해 구성하는 방법으로, 인덱스 타입에 대한 mappings 작성 시 문서 내 nested 타입을 정의하고 데이터 비정규화를 위한 필드를 선언합니다.

parent-child 타입을 예로 들면, parent 타입의 mappings에 child 타입의 필드를 생성하고 child 타입 필드를 nested 타입으로 정의한 후 child 타입의 필드를 추가하면 됩니다. 결국 두 타입을 하나의 타입으로 합친 것입니다.

이 기능을 사용하려면 일반 쿼리 API로는 사용할 수 없고, nested 쿼리/필터 API를 사용해야 합니다.

NOTE API 사용 시 주의 사항

nested path에 대한 조건 지정이 아니면 일반 쿼리 또는 필터 API를 사용해야 합니다.

앞에서 정의한 parent-child 모델을 nested 모델로 재설계하겠습니다. 즉, 두 개의 타입을 하나의 타입으로 구성하고 child 타입을 nested 타입으로 설정합니다.

buyer + buyer_item = purchase_history

purchase_history는 1:N 관계의 문서 구조를 하나의 문서 구조 안에 담고 있습니다.

nested 인덱스 생성

nested 인덱스 구성 시 settings는 일반적인 인덱스 settings와 동일하며,

mappings 구성 시 선언 방법이 달라지므로 다음 mappings 구성은 확인합니다(settings 설정 정보는 제공된 [소스 코드](#)⁰⁸를 참고하기 바랍니다).

child 타입의 buyer_item이 buyer_items로 바뀌었고 nested 타입으로 선언된 것을 확인할 수 있습니다.

[Nested 인덱스 생성 – mappings]

```
"mappings" : {  
    "purchase_history" : {  
        ...중략...  
        "properties" : {  
            ...중략...  
            "buyer_items" : {  
                "type" : "nested",  
                "properties" : {  
                    ...중략...  
                }  
            }  
        }  
    }  
}
```

nested 타입을 갖는 인덱스 생성은 코드에서 보는 것과 같이 일반적인 인덱스 생성과 다르지 않습니다.

[Nested 인덱스 생성 예제 코드 – JoinNestedTest.java]

```
setting = Operators.readfile("schema/join_nested_settings.json");  
mapping[0] = Operators.readfile("schema/join_nested_mappings.json");  
try {  
    client.admin().indices().delete(new DeleteIndexRequest("join_nested")).actionGet();  
} catch (Exception e) {  
} finally {  
}  
  
CreateIndexResponse createIndexResponse = client.admin().indices()  
    .prepareCreate("join_nested")  
    .setSettings(setting)  
    .addMapping("purchase_history", mapping[0])  
    .execute()
```

08 <http://bit.ly/1w1NcMw>

```
.actionGet();
```

Nested 타입 도큐먼트

_index, _type은 일반 도큐먼트 등록 구조와 동일하고, 도큐먼트 구조 역시 buyer_items를 제외하고는 일반 mappings 설정과 동일합니다. buyer_items는 nested로 선언했으므로 object 배열 리스트 형태로 데이터를 구성합니다.

[Nested 타입 도큐먼트 구조 – join_nested.json]

```
{ "index" : { "_index" : "join_nested", "_type" : "purchase_history" } }
{ "buyer_id" : 1,"buyer_login_id" : "jjeong","buyer_name" : "헨리","buyer_sex" : "M","buyer_phone" : "01019931974", "buyer_items" : [ {"item_id" : 1,"item_name" : "스캇 자전거","purchase_date" : "20140611121212","item_price" : 750000,"order_price" : 600000}, {"item_id" : 2,"item_name" : "자전거 트레일러","purchase_date" : "20140612121212","item_price" : 340000,"order_price" : 195000} ] }
```

이해를 돋기 위해 다음 Java 자료 구조를 살펴보겠습니다. 여기서 BuyerItem Model은 buyer_item 타입의 필드에 해당하는 setter/getter로 구성된 클래스입니다.

[Nested 타입 도큐먼트의 Java 자료 구조 예]

```
ArrayList<Object> nestedDoc = new ArrayList<Object>();
ArrayList<BuyerItemModel> nestedDoc = new ArrayList<BuyerItemModel>();
```

REST API로 등록합니다.

[Nested 타입 데이터 등록]

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

Nested 쿼리

일반 Query DSL과 다르게 nested로 시작해야 하고, path 필드를 지정해야 합니다.

니다. path 값은 nested 타입으로 선언한 필드를 등록합니다. nested 쿼리는 score_mode를 지원하며 다음 세 가지 모드를 사용할 수 있습니다.

- **avg** : 모든 child 문서의 score 평균
- **sum** : 모든 child 문서의 score 합계
- **max** : 모든 child 문서의 score 중 제일 큰 값

검색 질의를 하기 위한 query 영역에서는 dot()을 이용해서 nested 필드를 선언해야 합니다.

[Nested 쿼리의 Query DSL 코드 – JoinNestedTest.java]

```
"query": {  
    "nested" : {  
        "path" : "buyer_items", "score_mode" : "avg",  
        "query" : {  
            "term": {  
                "buyer_items.item_name": {  
                    "value": "자전거"  
                }  
            }  
        }  
    }  
}
```

Query DSL 예제를 그대로 Java API로 변환하였습니다.

[Nested 쿼리의 Java API 코드 – JoinNestedTest.java]

```
String path = "buyer_items";  
TermQueryBuilder termQueryBuilder = new TermQueryBuilder("buyer_items.item_name", "자전  
거");  
NestedQueryBuilder nestedQueryBuilder = new NestedQueryBuilder(path, termQueryBuilder);  
nestedQueryBuilder.scoreMode("avg");  
  
String searchResult = Operators.executeQuery(settings, client, nestedQueryBuilder, "join_  
nested");
```

parent-child 모델과 달리 nested path를 통해서 검색을 수행하였으나 결과에 모든 문서 정보가 포함됩니다. 따라서 nested 모델은 관계형 데이터를 모두 가져와 사용할 때 유용합니다.

Nested 쿼리 결과

```
"hits" : {  
    "total" : 2,  
    "max_score" : 1.287682,  
    "hits" : [ {  
        "_index" : "join_nested",  
        "_type" : "purchase_history",  
        "_id" : "2",  
        "_score" : 1.287682,  
        "_source":{ "buyer_id" : 2,"buyer_login_id" : "atie","buyer_name" : "아띠","buyer_sex" : "F","buyer_phone" : "01020011982", "buyer_items" : [ {"item_id" : 2,"item_name" : "자전거 트레일러","purchase_date" : "20140611121212","item_price" : 340000,"order_price" : 195000}, {"item_id" : 3,"item_name" : "비싼 자전거","purchase_date" : "20130610121212","item_price" : 7500000,"order_price" : 1000000} ] }  
    }, {  
        "_index" : "join_nested",  
        "_type" : "purchase_history",  
        "_id" : "1",  
        "_score" : 1.0,  
        "_source":{ "buyer_id" : 1,"buyer_login_id" : "jjeong","buyer_name" : "헨리","buyer_sex" : "M","buyer_phone" : "01019931974", "buyer_items" : [ {"item_id" : 1,"item_name" : "스캇 자전거","purchase_date" : "20140611121212","item_price" : 750000,"order_price" : 600000}, {"item_id" : 2,"item_name" : "자전거 트레일러","purchase_date" : "20140612121212","item_price" : 340000,"order_price" : 195000} ] }  
    } ]  
}
```

Nested 필터

Nested 쿼리와 달리 join 속성을 가지며, 이것을 통해 nested path에 대한 조인 연산을 수행합니다. join 속성의 값이 'false'면 nested 필터 수행 결과가 반환되지 않습니다.

[Nested 필터의 Query DSL 코드 – JoinNestedTest.java]

```
"filtered" : {  
    "query" : {  
        "match_all" : { }  
    },  
    "filter" : {  
        "nested" : {  
            "filter" : {  
                "term" : {  
                    "buyer_items.item_id" : 2  
                }  
            },  
            "join" : true,  
            "path" : "buyer_items"  
        }  
    }  
}
```

Query DSL 예제를 그대로 Java API로 변환하였습니다.

[Nested 필터의 Java API 코드 – JoinNestedTest.java]

```
String path = "buyer_items";  
TermFilterBuilder termFilterBuilder = new TermFilterBuilder("buyer_items.item_id", 2);  
NestedFilterBuilder nestedFilterBuilder = new NestedFilterBuilder(path,  
termFilterBuilder);  
FilteredQueryBuilder filteredQueryBuilder = new FilteredQueryBuilder(new  
MatchAllQueryBuilder(), nestedFilterBuilder);  
nestedFilterBuilder.join(true);  
  
String searchResult = Operators.executeQuery(settings, client, filteredQueryBuilder,  
"join_nested");
```

Nested 쿼리 결과와 달리 _score 값에 대한 연산을 수행하지 않아서 모든 값은 '1.0'이 됩니다. 이에 관한 자세한 설명은 '[7.3.2 검색 튜닝](#)'에서 다루겠습니다.

Nested 필터 결과

```
"hits" : {  
    "total" : 2,
```

```
"max_score" : 1.0,  
"hits" : [ {  
    "_index" : "join_nested",  
    "_type" : "purchase_history",  
    "_id" : "1",  
    "_score" : 1.0,  
    "_source":{ "buyer_id" : 1,"buyer_login_id" : "jjeong","buyer_name" : "헨  
리","buyer_sex" : "M","buyer_phone" : "01019931974", "buyer_items" : [ {"item_id" : 1,"item_  
name" : "스갓 자전거","purchase_date" : "20140611121212","item_price" : 750000,"order_  
price" : 600000}, {"item_id" : 2,"item_name" : "자전거 트레일러","purchase_date" :  
"20140612121212","item_price" : 340000,"order_price" : 195000} ] }  
},  
...중략...  
]  
}
```

1.4 River

River는 클러스터Cluster⁰⁹로 색인 데이터를 밀어 넣어주는 플러그인Plugin 형태의 서비스를 의미합니다. Elasticsearch에서 매우 다양한 River 플러그인을 제공하므로 여기서는 River 기본 개념을 이해하고 기능 테스트를 위한 JDBC River¹⁰를 구현하는 방법을 살펴보겠습니다.

1.4.1 JDBC River 동작의 이해

JDBC River가 Elasticsearch에 설치되어 있으면 해당 River에 접속할 RDBMS의 정보와 색인할 데이터를 가져오기 위한 쿼리를 작성하여 등록합니다. 등록이 완료되면 등록된 스케줄에 따라 주기적으로 쿼리를 실행하고 색인 데이터

⁰⁹ standalone으로 동작하는 여러 노드를 하나의 그룹으로 묶어서 데이터의 분산과 공유를 할 수 있게 서비스를 구성하는 것을 말한다.

¹⁰ <https://github.com/jprante/elasticsearch-river-jdbc>

를 가져와 색인 작업을 수행합니다. Elasticsearch 클러스터 환경에서 River는 하나의 노드¹¹에서만 동작하고, 해당 노드에서 장애가 발생하면 Failover 기능이 동작합니다.

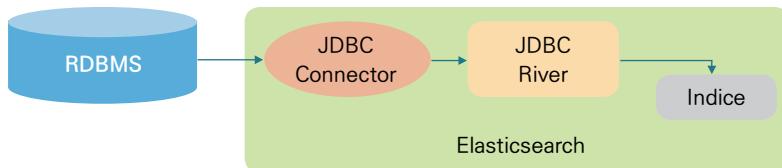
NOTE Failover

Elasticsearch에서는 마스터 노드^{Master Node}에 장애가 발생하면 마스터 노드 선출 과정을 거쳐 장애에 대응합니다. 이와 비슷하게 River 역시 장애가 발생하면 자동으로 다른 노드에서 장애를 감지하고 장애에 대응하는데, 이 과정이 Failover 기능입니다.

동작 순서를 정리하면 다음과 같습니다.

- JDBC River 설치
- JDBC 접속 정보 설정
- 색인 데이터 페치^{Fetch} 쿼리 설정
- 데이터 페치 주기 설정
- 기타 설정

그림 1-11 JDBC River 아키텍처



- **RDBMS** : 색인 원본 데이터가 저장된 저장소
- **JDBC Connector** : RDBMS와 연결하기 위한 JDBC 드라이버
- **JDBC River** : Elasticsearch에서 JDBC Connector를 이용하여 RDBMS의 데이터를 주기적으로 풀링^{Pulling}하고, 클러스터로 색인하여 저장하는 플러그인 형태의 서비스
- **Indice** : JDBC River에 의해 풀링된 데이터를 색인 및 질의하기 위한 저장소

¹¹ Elasticsearch를 구성하는 하나의 서버 또는 데몬으로, 독립적으로 동작할 수 있다.

1.4.2 JDBC River 설치 전 준비작업

플러그인 설치 및 기능 점검에 앞서 MySQL 서버가 설치되고 실행되어야 합니다. 그리고 풀링할 데이터도 등록되어야 합니다(MySQL 서버 설치 및 실행 방법은 별도 설명하지 않으므로 각자 정보를 찾아서 참고하길 바랍니다).

하나의 테이블에서 상품의 가격 정보 변동을 색인하는 간단한 예제를 살펴보겠습니다. 다음 테이블은 쇼핑몰 상품의 가격 정보 변동을 반영하기 위해 충분 색인¹² 용 데이터를 적재합니다.

그림 1-12 JDBC River용 테이블 구조

TBL_ITEM_INCREMENT	
!	item_id BIGINT
◆	item_code VARCHAR(45)
◆	item_name VARCHAR(45)
◆	price BIGINT
◆	regdate DATETIME
◆	ts TIMESTAMP
◆	fetch_flag CHAR(1)
Indexes	
PRIMARY	
item_id_UNIQUE	
regdate_idx	
fetch_idx	

표 1-3 JDBC River용 테이블 상세

필드명	필드 속성값	설명
item_id	-	증분 색인을 위한 기본키
item_code	-	상품의 기본키에 해당하는 고유 코드

¹² 기존 문서 정보가 변경되어 반영해야 하거나 일부 신규 데이터가 추가되어 등록해야 할 때 색인하는 것을 의미 합니다.

필드명	필드 속성값	설명
item_name	-	상품명
price	-	상품 가격
regdate	-	등록일
ts	-	Timestamp
fetch_flag	'F' or 'T'	증분 색인 여부(F : 색인 전, T : 색인 후)

이 예제는 MySQL 설치 시 기본 생성되는 test 데이터베이스에 테이블을 생성하였으나 각자 환경에 맞춰 생성하면 됩니다.

[테이블 생성 스크립트 – river_jdbc_mysql.json]

```
CREATE TABLE IF NOT EXISTS `test`.`TBL_ITEM_INCREMENT` (
    `item_id` BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
    `item_code` VARCHAR(45) NOT NULL,
    `item_name` VARCHAR(45) NOT NULL,
    `price` BIGINT NOT NULL,
    `regdate` DATETIME NOT NULL,
    `ts` TIMESTAMP NOT NULL,
    `fetch_flag` CHAR(1) NOT NULL DEFAULT 'F',
    PRIMARY KEY (`item_id`),
    UNIQUE INDEX `item_id_UNIQUE` (`item_id` ASC),
    INDEX `regdate_idx` (`regdate` ASC),
    INDEX `fetch_idx` (`ts` ASC, `fetch_flag` ASC)
DEFAULT CHARACTER SET UTF8 COLLATE UTF8_GENERAL_CI
ENGINE = InnoDB;
```

Insert 문으로 데이터를 등록합니다.

[샘플 데이터 등록 – river_jdbc_mysql.json]

```
INSERT INTO tbl_item_increment(`item_code`, `item_name`, `price`, `regdate`)
VALUES('AA001', '비싼 자전거', 10000000, now());
INSERT INTO tbl_item_increment(`item_code`, `item_name`, `price`, `regdate`)
VALUES('AA002', '싼 자전거', 10000000, now());
INSERT INTO tbl_item_increment(`item_code`, `item_name`, `price`, `regdate`)
VALUES('AA003', '최신 자전거', 1000000, now());
...중략...
```

1.4.3 JDBC River 설치

MySQL 서버에 테이블 생성과 데이터 등록이 완료되면 JDBC River 플러그인을 설치합니다.

MySQL JDBC River 플러그인 설치

기존에 설치된 플러그인이 있으면 삭제한 후 설치합니다.

삭제 후 설치

```
$ bin/plugin --remove jdbc  
$ bin/plugin --install jdbc --url http://xbib.org/repository/org/xbib/elasticsearch/plugin/  
elasticsearch-river-jdbc/1.3.4.4/elasticsearch-river-jdbc-1.3.4.4-plugin.zip
```

MySQL JDBC Connector 설치

다운로드

```
$ curl -o mysql-connector-java-5.1.31.zip -L 'http://dev.mysql.com/get/Downloads/  
Connector-J/mysql-connector-java-5.1.31.zip/from/http://cdn.mysql.com/'
```

압축 해제와 복사

```
$ unzip mysql-connector-java-5.1.31.zip  
$ cp mysql-connector-java-5.1.31/mysql-connector-java-5.1.31-bin.jar ..../elasticsearch/  
plugins/jdbc/
```

압축 파일은 임의의 위치에서 해제한 후 Elasticsearch 설치 위치의 JDBC River 설치 경로(`plugins/jdbc`)로 복사합니다.

ElasticSearch 재시작

JDBC River가 정상적으로 동작하려면 플러그인과 커넥터를 설치한 후 반드시 재시작해야 합니다. 클러스터 환경일 때는 모든 노드에 동일하게 적용한 후 재시작해야 합니다.

1.4.4 JDBC River Indice 구성

JDBC River를 이용해서 DB 소스를 가져와서 색인할 대상 indice를 생성합니다. 여기서 사용하는 예제는 하나의 테이블에서 상품의 가격 정보 변동을 색인하므로 같은 문서 구조로 매팅 정보를 작성하겠습니다(전체 정보는 제공된 소스 코드를 참고하기 바랍니다).

```
[TBL_ITEM_INCREMENT indice settings – river_jdbc_mysql.json] ——————  
{  
    "number_of_shards" : 3,  
    "number_of_replicas" : 0,  
    "index" : {  
        ...중략...  
    }  
}
```

regdate 필드는 검색 질의할 때 다양한 조건으로 활용하기 위해 멀티 필드로 구성합니다.

```
[TBL_ITEM_INCREMENT type mappings – river_jdbc_mysql.json] ——————  
{  
    "tbl_item_increment" : {  
        ...중략...  
        "properties" : {  
            "item_id" : {"type" : "long", "store" : "no", "index" : "not_analyzed", "index_options" : "docs", "ignore_malformed" : true, "include_in_all" : false},  
            ...중략...  
            "regdate" : {  
                "type" : "date", "format" : "yyyyMMddHHmmss", "store" : "no", "index" : "not_analyzed", "index_options" : "docs", "ignore_malformed" : true, "include_in_all" : false,  
                "fields" : {  
                    "regdate_search" : {"type" : "long", "store" : "no", "index" : "not_analyzed", "index_options" : "docs", "ignore_malformed" : true, "include_in_all" : false}  
                }  
            }  
        }  
    }
```

```
    }  
}
```

MySQL용 JDBC River 인덱스명은 `river_jdbc_mysql`로 생성하고, MySQL에서 생성한 테이블명과 일치시키기 위해 `mappings` 생성 시 인덱스 타입명은 `tbl_item_increment`로 설정합니다.

[JDBC River indice 생성 예제 코드 – `RiverJdbcTest.java`]

```
String setting = "";  
String[] mapping = new String[1];  
  
setting = Operators.readFile("schema/river_jdbc_settings.json");  
mapping[0] = Operators.readFile("schema/river_jdbc_mappings.json");  
  
try {  
    client.admin().indices().delete(new DeleteIndexRequest("river_jdbc_mysql"));  
    actionGet();  
} catch (Exception e) {  
} finally {  
}  
  
CreateIndexResponse createIndexResponse = client.admin().indices()  
    .prepareCreate("river_jdbc_mysql")  
    .setSettings(setting)  
    .addMapping("tbl_item_increment", mapping[0])  
    .execute()  
    .actionGet();  
  
client.close();
```

1.4.5 JDBC River 등록

JDBC River의 핵심은 River의 등록입니다. River의 연결 정보와 동작에 관한 모든 내용은 등록할 때 구성하므로 등록 스크립트를 잘 작성해야 합니다. 기본으로 다음 정보를 등록합니다.

표 1-4 River 파라미터

파라미터	설명
strategy	River 동작 전략 정의(simple, column)
schedule	계획된 작업 실행 일정 정의
threadpoolsize	schedule 실행을 위한 스레드 크기
interval	River 간 실행 간격
max_bulk_actions	개별 bulk 요청의 최대 색인 크기
max_concurrent_bulk_requests	동시 실행 가능한 bulk 요청 최대 크기
max_bulk_volume	bulk 요청의 최대 크기
max_request_wait	bulk 요청 후 응답 대기까지 최대 시간
flush_interval	bulk 실행 후 색인 문서에 대한 flush 간격

표 1-5 JDBC 파라미터

파라미터	설명
url	JDBC 드라이버 URL
user	JDBC DB user
password	JDBC DB password
sql	SQL 구문
statement	insert/update문 실행 제어
write	CallableStatement 실행 제어
callable	SQL 구문에 대한 파라미터 할당
parameter	언어 설정
locale	시간대 설정
timezone	숫자 값에 대한 rounding 모드 설정
rounding	숫자 값에 대한 precision 설정
scale	AutoCommit 설정
autocommit	ResultSet의 페치 크기 설정
fetchsize	fetchsize에 대한 최대 row 값 설정
max_rows	DB 재연결 시 최대 재시도 횟수 설정
max_retries	DB 연결 시도 시 대기하는 최대 시간 설정
max_retries_wait	

파라미터	설명
resultset_type	java.sql.ResultSet에서 제공하는 TYPE_ 정의
resultset_concurrency	java.sql.ResultSet에서 제공하는 CONCUR_ 정의
ignore_null_values	NULL 값에 대한 무시 여부
prepare_database_metadata	DB 메타 데이터 사용 여부
prepare_resultset_metadata	Resultset 메타 데이터 사용 여부
column_name_map	DB column명에 대한 별칭 설정
query_timeout	SQL 구문 timeout 설정
connection_properties	Connection properties 설정
index	ElasticSearch 내 인덱스명
type	ElasticSearch 내 인덱스의 타입명
index_settings	ElasticSearch 내 인덱스 setting 정보
type_mapping	ElasticSearch 내 인덱스의 타입 mapping 정보

다음은 TBL_ITEM_INCREMENT 테이블에서 반영 전인 데이터를 2개씩 가져와서 기존 인덱스에 충분 색인하는 예제입니다. 데이터 추출은 1분 단위로 이루어집니다.

[JDBC River 메타 데이터의 DSL 설정 - 원본 데이터 추출]

```
{
  "type" : "jdbc",
  "jdbc" : {
    "url" : "jdbc:mysql://localhost:3306/test",
    "user" : "root",
    "password" : "",
    "sql" : [
      {
        "statement" : "SELECT item_id, item_code, item_name, price, regdate FROM TBL_ITEM_INCREMENT WHERE fetch_flag = ? LIMIT 0, 2",
        "parameter" : ["F"],
        "callable" : false
      }
    ],
    "index" : "river_jdbc_mysql",
    "type" : "tbl_item_increment",
  }
}
```

```
        "schedule" : "0 0-59 0-23 ? * *"
    }
}
```

JSON string 형태로 구성된 것을 Java API를 이용해서 구현하였습니다. 여기서 River의 메타 데이터는 _river라는 인덱스에 등록되고, 메타 데이터는 지정한 타입으로 등록됩니다. 메타 데이터의 등록 상태는 다음 REST API로 확인할 수 있습니다.

[JDBC River 메타 데이터의 Java API 설정 – 원본 데이터 추출]

```
IndexRequestBuilder requestBuilder;
IndexResponse response;

XContentBuilder json = jsonBuilder().startObject();
    json.field("type", "jdbc");
    json.field("jdbc");
    json.startObject();
        json.field("url", "jdbc:mysql://localhost:3306/test");
        json.field("user", "root");
        json.field("password", "");
        json.field("sql");
        json.startArray();
            json.startObject();
                json.field("statement", "SELECT item_id, item_code, item_name, price,
regdate FROM TBL_ITEM_INCREMENT WHERE fetch_flag = ? LIMIT 0, 2");
            json.field("parameter");
            json.startArray();
                json.value("F");
            json.endArray();
            json.field("callable", false);
        json.endObject();
    json.endArray();
    json.field("index", "river_jdbc_mysql");
    json.field("type", "tbl_item_increment");
    json.field("schedule", "0 0-59 0-23 ? * *");

    json.endObject();
json.endObject();
```

```
requestBuilder = client.prepareIndex("_river", "jdbc_mysql_fetch");
response = requestBuilder.setId("_meta")
    .setSource(json)
    .execute()
    .actionGet();
```

정상적으로 등록되지 않으면 에러 메시지를 보여줍니다.

```
// http://localhost:9200/_river/my_river_type/_status
$ curl -XGET http://localhost:9200/_river/jdbc_mysql_fetch/_status?pretty=true
```

메타 데이터는 다음 REST API로 삭제할 수 있습니다.

```
// http://localhost:9200/_river/my_river_type
$ curl -XDELETE http://localhost:9200/_river/jdbc_mysql_fetch
```

다음은 앞에서 폐치해 온 원본 데이터를 색인 반영한 후 다시 폐치되지 않도록 플래그^{Flag}를 변경하는 예제입니다. 두 예제는 1분 단위로 폐치하고 update가 적용됩니다.

[JDBC River 메타 데이터의 DSL 설정 – 원본 데이터 반영]

```
{
  "type" : "jdbc",
  "jdbc" : {
    "url" : "jdbc:mysql://localhost:3306/test",
    "user" : "root",
    "password" : "",
    "sql" : [
      {
        "statement" : "UPDATE (SELECT * FROM TBL_ITEM_INCREMENT WHERE fetch_flag=? LIMIT 0,2) src, TBL_ITEM_INCREMENT dest SET dest.fetch_flag=? WHERE dest.item_id=tbl.item_id",
        "parameter" : ["F", "T"],
        "callable" : false,
        "write" : true
      }
    ]
  }
}
```

```

        }
    ],
    "index" : "river_jdbc_mysql",
    "type" : "tbl_item_increment",
    "schedule" : "0 0-59 0-23 ? * *"
}
}

```

메타 데이터에 지정한 jdbc_mysql_update 타입으로 등록됩니다. 메타 데이터 등록 상태는 다음 REST API로 확인할 수 있습니다.

[JDBC River 메타 데이터의 Java API 설정 - 원본 데이터 반영]

```

IndexRequestBuilder requestBuilder;
IndexResponse response;

XContentBuilder json = jsonBuilder().startObject();
    json.field("type", "jdbc");
    json.field("jdbc");
    json.startObject();
        json.field("url", "jdbc:mysql://localhost:3306/test");
        json.field("user", "root");
        json.field("password", "");
        json.field("sql");
        json.startArray();
            json.startObject();
                json.field("statement", "UPDATE (SELECT * FROM TBL_ITEM_INCREMENT WHERE
fetch_flag=? LIMIT 0,2) src, TBL_ITEM_INCREMENT dest SET dest.fetch_flag=? WHERE dest.
item_id=src.item_id");
                    json.field("parameter");
                    json.startArray();
                        json.value("F");
                        json.value("T");
                    json.endArray();
                    json.field("callable", false);
                    json.field("write", true);
            json.endObject();
    json.endArray();
    json.field("index", "river_jdbc_mysql");
    json.field("type", "tbl_item_increment");
}

```

```
        json.field("schedule", "0 0-59 0-23 ? * *");

    json.endObject();
json.endObject();

requestBuilder = client.prepareIndex("_river", "jdbc_mysql_update");
response = requestBuilder.setId("_meta")
    .setSource(json)
    .execute()
    .actionGet();
```

정상적으로 등록되지 않으면 에러 메시지를 보여줍니다.

```
// http://localhost:9200/_river/my_river_type/_status
$ curl -XGET http://localhost:9200/_river/jdbc_mysql_update/_status?pretty=true
```

메타 데이터는 다음 REST API로 삭제할 수 있습니다.

```
// http://localhost:9200/_river/my_river_type
$ curl -XDELETE http://localhost:9200/_river/jdbc_mysql_update
```

1.4.6 JDBC River 실행

JDBC River가 정상적으로 등록되면 등록한 schedule에 의해 자동으로 실행됩니다. Config/loggin.yml에서 es.logger.level을 INFO에서 DEBUG로 변경한 후 Elasticsearch 데몬을 재시작하면 로그 파일에 실행 로그가 쌓이는 것을 확인할 수 있습니다.

Config/loggin.yml 파일 설정

```
# before
es.logger.level: INFO

# after
es.logger.level: DEBUG
```

JDBC River 정상 실행 로그

```
[2014-11-12 18:31:00,003][DEBUG][river.jdbc.RiverThread] river flow org.xbib.elasticsearch.river.jdbc.strategy.simple.SimpleRiverFlow@5c905111 thread is starting  
...중략...  
[2014-11-12 18:31:00,021][DEBUG][cluster.service] [hanbit] processing [post_river_state[jdbc_mysql_update]]: execute  
...중략...  
[2014-11-12 18:31:00,027][DEBUG][cluster.service] [hanbit] processing [post_river_state[jdbc_mysql_fetch]]: execute  
...중략...  
[2014-11-12 18:31:00,105][DEBUG][river.jdbc.RiverThread] shutting down metrics thread scheduler  
[2014-11-12 18:31:00,105][DEBUG][river.jdbc.RiverThread] shutting down suspension thread scheduler  
[2014-11-12 18:31:00,105][DEBUG][river.jdbc.RiverThread] river flow org.xbib.elasticsearch.river.jdbc.strategy.simple.SimpleRiverFlow@6ba6291 thread is finished
```

1.4.7 JDBC River에서 REST API 이용하기

River를 사용하다 보면 정상적으로 등록되었는지 확인하거나 실행을 중지 또는 재시작해야 할 경우가 생기는데, 이럴 때 다음 REST API를 이용하면 쉽게 운영할 수 있습니다.

다음은 등록한 River에 대한 상태 정보를 보여주고 정상 동작을 확인합니다.

REST API _state

```
$ curl -XGET http://localhost:9200/_river/jdbc/{rivername}/_state
```

현재 등록된 River의 실행을 중지합니다.

REST API _suspend

```
$ curl -XPOST http://localhost:9200/_river/jdbc/{rivername}/_suspend -d "{\"rivername\":\"{my_jdbc_river}\")"
```

실행 중지된 River를 다시 실행합니다.

REST API _resume

```
$ curl -XPOST http://localhost:9200/_river/jdbc/{rivername}/_resume -d "{\"rivername\":\"\"\"{my_jdbc_river}\"\"\""}"
```

실행 중인 River를 강제로 중지합니다.

REST API _abort

```
$ curl -XPOST http://localhost:9200/_river/jdbc/{rivername}/_abort -d "{\"rivername\":\"\"\"{my_jdbc_river}\"\"\""}"
```

지정한 River를 실행합니다.

REST API _run

```
$ curl -XPOST http://localhost:9200/_river/jdbc/{rivername}/_run -d "{\"rivername\":\"\"\"{my_jdbc_river}\"\"\""}"
```

_suspend, _resume, _abort, _run은 rivername이라는 변수로 등록한 River명을 넘겨야 정상적으로 동작하고, 그렇지 않으면 다음 오류 메시지가 출력됩니다.

오류 메시지

```
{"error":"NullPointerException[null]","status":500}
```

1.5 정리

검색 확장 기능은 앞의 예제에서 살펴보았듯이 검색 서비스와 그 외 다양한 영역의 서비스에서 활용할 수 있습니다. 예를 들어, 자동 완성은 오타 교정, 추천 기능으로 활용할 수 있고, percolator는 실시간 모니터링, 데이터 검증 등에서 활용됩니다. 또한, Join과 River는 구조적인 기능 구현의 편리성을 제공하므로 이 기능들을 응용하여 서비스에 접목한다면 다양한 분야에 활용할 수 있습니다.

검색 데이터 분석

Elasticsearch에서는 이미 색인된 데이터를 기반으로 다양한 분석 API를 제공합니다. 루씬Lucene에서는 facet이라는 데이터 분석 API를 제공하고 Elasticsearch에서도 최근까지 이 기능을 제공했습니다. 하지만 실시간 분석에 대한 다양한 요구가 늘어나면서 Elasticsearch에서는 facet 기능을 참조하여 aggregation이라는 새로운 대체 API를 지원하고 있습니다.

aggregation은 질의 조건과 일치하는 문서에 대한 결과와 정보를 실시간으로 분석하는 도구로, 기본 구조는 다음과 같습니다.

```
"aggregations" : {  
    "<aggregation_name>" : {  
        "<aggregation_type>" : {  
            <aggregation_body>  
        }  
        [ , "aggregations" : { [ <sub_aggregation> ]+ } ]?  
    }  
    [ , "<aggregation_name_2>" : { ... } ]*  
}
```

- <aggregation_name> : aggregation 실행 시 지정한 결과의 이름
- <aggregation_type> : aggregation 타입명
- <aggregation_body> : aggregation 실행 결과
- <sub_aggregation> : sub aggregation 실행 결과

aggregation마다 각각의 사용 목적과 결과가 있지만 Elasticsearch에서 제공하는 aggregation API에는 아직 실험적인 것도 있어서 여기에서는 많이 사용하는 bucket과 metric aggregation을 알아보겠습니다.

이 장에서 사용할 예제의 기본 인덱스는 다음과 같이 생성합니다.

- **인덱스명** : aggregations
- **타입명** : transactions

표 2-1 필드 구성

필드명	설명
item_code	상품코드
item_name	상품명
item_category	상품분류
buyer_id	구매자아이디
buyer_gender	구매자성별
buyer_location	구매위치
buyer_country	구매국가
payment_price	결제금액
payment_type	결제유형
payment_time	결제시간

2.1 Bucket Aggregation

bucket은 하나의 key 필드와 같은 문서 특징을 갖는 집계로, 이 집계를 통합하는 과정이 bucket aggregation입니다. bucket aggregation에 대한 이해를 돋기 위해 간단히 term aggregation 실행 결과를 살펴보겠습니다.

term aggregation 실행 결과

```
"aggregations" : {  
    "aggs_result" : {
```

```
...중략...
"buckets" : [ {
    "key" : "card",
    "doc_count" : 6
}
...중략...
]
}
}
```

하나의 bucket은 key 필드와 doc_count 필드로 구성되고, key 필드는 ‘card’라는 색인어를 기준으로 문서가 분류되며, doc_count 필드는 ‘card’라는 색인어를 가진 문서의 수가 기록되어 있습니다.

bucket aggregation에는 다음과 같은 종류가 있는데, 이 중에서 일부를 좀 더 자세히 살펴보겠습니다. 실행 가능한 전체 소스 코드는 제공된 코드를 참고하기 바랍니다.

global, filter, filters, missing, nested, reverse nested, children, terms, significant terms, range, date range, ip range, histogram, date histogram, geo distance, geo hash grid

2.1.1 Global Aggregation

global aggregation은 다른 aggregation API와 기본적인 검색 질의 방식은 같지만, 결과를 얻으려면 반드시 sub aggregation을 등록해서 실행해야 합니다. sub aggregation은 검색 질의 결과 문서가 아닌 전체 문서를 대상으로 결과를 생성하므로 검색 질의 조건과 상관없이 전체 문서를 대상으로 분석하고 싶을 때 유용한 API입니다.

다음은 구매국가가 한국인 구매내역을 조회하고 이 구매내역의 평균 구매금액을 구하는 예제로, global aggregation은 선언만 합니다.

[Query DSL]

```
{  
    "query" : {  
        "term" : { "buyer_country" : "KR" }  
    },  
    "aggs" : {  
        "aggs_result" : {  
            "global" : {},  
            "aggs" : {  
                "sub_aggs_result" : { "avg" : { "field" : "payment_price" } }  
            }  
        }  
    }  
}
```

[Java API]

```
GlobalBuilder aggsBuilder = AggregationBuilders.global("aggs_result");  
AvgBuilder subAggsBuilder = AggregationBuilders.avg("sub_aggs_result");  
subAggsBuilder.field("payment_price");  
aggsBuilder.subAggregation(subAggsBuilder);  
  
String searchResult = Operators.executeAggregation(settings, client, new  
TermQueryBuilder("buyer_country", "KR"), aggsBuilder, "aggregations", "transactions");
```

global aggregation 결과는 doc_count라는 단일 집계 값만 가지게 되고, 실제 수행하려는 aggregation(한국에서 발생한 구매내역의 평균 구매금액) 결과는 sub_aggs_result에서 얻게 됩니다.

실행 결과

```
"aggregations" : {  
    "aggs_result" : {  
        "doc_count" : 21,  
        "sub_aggs_result" : {  
            "value" : 114580.47619047618  
        }  
    }  
}
```

2.1.2 Filter Aggregation

기본적인 검색 질의 결과를 대상으로 필터 조건을 추가하여 조건에 일치하는 문서들을 분석하는 API입니다. aggregation 작성 시 필터 질의와 sub aggregation을 작성해야 합니다. 이 API는 검색 질의 결과에 추가로 특정 조건에 대한 분석이 필요할 때 유용합니다.

NOTE Filter

Elasticsearch에서 제공하는 필터의 방식을 알아보겠습니다.

- **filtered** : 질의 수행 시 검색과 분석 결과 모두에 영향을 미칩니다.
- **filter** : 질의 수행 시 검색 결과를 바탕으로 분석 결과에만 영향을 미칩니다.
- **post filter** : 질의 수행 시 검색 결과에만 영향을 미치며 분석 결과에는 영향을 미치지 않습니다.

다음은 전체 문서(구매내역) 중 특정 시간 이후로 결제된 문서의 평균 결제금액을 계산하는 예제입니다. 범위 조건에서 `from`을 사용하면 기본으로 범위값에 대한 `equals`가 반영된다는 점에서 `gt01`와 차이가 있습니다(`to`도 동일합니다).

[Query DSL]

```
"aggs" : {  
    "aggs_result" : {  
        "filter" : { "range" : { "payment_time" : { "from" : 20140619163000 } } },  
        "aggs" : {  
            "sub_aggs_result" : { "avg" : { "field" : "payment_price" } }  
        }  
    }  
}
```

[Java API]

```
RangeFilterBuilder filterBuilder = FilterBuilders.rangeFilter("payment_time").  
from("20140619163000"); // default equals true  
// RangeFilterBuilder filterBuilder = FilterBuilders.rangeFilter("payment_time").
```

⁰¹ 표 6-8 range 쿼리 연산자 참고

```
gt("20140619163000");
FilterAggregationBuilder aggsBuilder = AggregationBuilders.filter("aggs_result");
AvgBuilder subAggsBuilder = AggregationBuilders.avg("sub_aggs_result");
subAggsBuilder.field("payment_price");

aggsBuilder.filter(filterBuilder);
aggsBuilder.subAggregation(subAggsBuilder);
```

필터 조건에 따라 doc_count가 3이 나온 것을 확인할 수 있습니다.

실행 결과

```
" aggregations" : {
    "aggs_result" : {
        "doc_count" : 3,
        "sub_aggs_result" : {
            "value" : 349000.0
        }
    }
}
```

2.1.3 Missing Aggregation

집계 문서에서 지정된 필드가 정의되지 않았거나 값이 'null'인 문서의 통계 수치를 반환하는 API로, 질의 조건의 결과에서 특정 필드의 값이 누락되었을 경우 이를 분석하기 위해 사용합니다.

다음은 구매정보 중 구매국가 정보가 누락된 문서를 분석하는 예제입니다.

[Query DSL]

```
"aggs" : {
    "aggs_result" : { "missing" : { "field" : "buyer_country" } }
```

[Java API – AggregationTest.java]

…중략…

```
MissingBuilder aggsBuilder = AggregationBuilders.missing("aggs_result");
```

```
aggsBuilder.field("buyer_country");
...중략...
```

등록된 데이터를 확인해 보면 임의로 한 문서에서 buyer_country 필드를 누락하고 등록하였으므로 전체 21개 문서 중 1개의 누락된 문서가 결과로 나옵니다.

실행 결과

```
"aggregations" : {
    "aggs_result" : {
        "doc_count" : 1
    }
}
```

2.1.4 Nested Aggregation

nested 타입을 가지는 문서의 분석을 지원하는 API로, 기본 검색 질의와 sub aggregation을 작성하여 사용합니다.

다음은 ‘1.3 Join’에서 구성한 인덱스와 문서를 대상으로 구매자의 구매상품 가격에 대한 평균값을 구하는 예제입니다.

[Query DSL]

```
"aggs" : {
    "aggs_result" : {
        "nested" : {
            "path" : "buyer_items"
        },
        "aggs" : {
            "sub_aggs_result" : { "avg" : { "field" : "buyer_items.item_price" } }
        }
    }
}
```

```

NestedBuilder aggsBuilder = AggregationBuilders.nested("aggs_result");
AvgBuilder subAggsBuilder = AggregationBuilders.avg("sub_aggs_result");
subAggsBuilder.field("buyer_items.item_price");

aggsBuilder.path("buyer_items");
aggsBuilder.subAggregation(subAggsBuilder);

String searchResult = Operators.executeAggregation(settings, client, new
MatchAllQueryBuilder(), aggsBuilder, "join_nested", "purchase_history");

```

이 예제에서 사용한 인덱스는 join_nested, 타입은 purchase_history입니다.

실행 결과

```

"aggregations" : {
  "aggs_result" : {
    "doc_count" : 4,
    "sub_aggs_result" : {
      "value" : 2232500.0
    }
  }
}

```

2.1.5 Reverse Nested Aggregation

nested aggregation과 같은 기능을 수행하고 nested 문서를 포함한 parent 도큐먼트를 분석하여 결과를 넘겨주는 역할을 하는 API입니다. 이 aggregation 은 nested aggregation의 결과를 바탕으로 결과를 만들기 때문에 반드시 nested aggregation 안에 선언해야 합니다.

다음은 구매자를 기준으로 구매상품 ID별 통계를 구하고, 구매상품별 구매자목록 을 추가로 구하는 예제입니다.

```

"aggs": {
  "aggs_result": {

```

```
"nested": {
    "path": "buyer_items"
},
"aggs": {
    "sub_aggs_result": {
        "terms": {
            "field": "buyer_items.item_id"
        },
        "aggs": {
            "rev_aggs_result": {
                "reverse_nested": {},
                "aggs": {
                    "rev_sub_aggs_result": {
                        "terms": {
                            "field": "buyer_login_id"
                        }
}

```

...중략...

[Java API]

```
NestedBuilder aggsBuilder = AggregationBuilders.nested("aggs_result");
TermsBuilder subAggsBuilder = AggregationBuilders.terms("sub_aggs_result");
ReverseNestedBuilder reverseAggsBuilder = AggregationBuilders.reverseNested("rev_aggs_
result");
TermsBuilder revSubAggsBuilder = AggregationBuilders.terms("rev_sub_aggs_result");

revSubAggsBuilder.field("buyer_login_id");
reverseAggsBuilder.subAggregation(revSubAggsBuilder);
subAggsBuilder.field("buyer_items.item_id");
subAggsBuilder.subAggregation(reverseAggsBuilder);
aggsBuilder.path("buyer_items");
aggsBuilder.subAggregation(subAggsBuilder);

String searchResult = Operators.executeAggregation(settings, client, new
MatchAllQueryBuilder(), aggsBuilder, "join_nested", "purchase_history");
```

실행 결과

```
"aggregations" : {
    "aggs_result" : {
        "doc_count" : 4,
```

```
"sub_aggs_result" : {
    "buckets" : [ {
        "key" : 2,
        "doc_count" : 2,
        "rev_aggs_result" : {
            "doc_count" : 2,
            "rev_sub_aggs_result" : {
                "buckets" : [ {
                    "key" : "atie",
                    "doc_count" : 1
                }, {
                    "key" : "jjeong",
                    "doc_count" : 1
                } ]
            }
        }
    },
    ...
    ...중략...
}
```

item_id가 2인 상품의 구매수량은 2개고, 이 2개 상품을 구매한 구매자의 ID 정 보가 reverse nested aggregation 결과로 보이는 것을 확인할 수 있습니다.

2.1.6 Terms Aggregation

집계 문서에서 추출된 색인어의 통계 정보를 반환하는 API로, 여러 개의 반환값 을 갖습니다. 어떤 특징이나 특성을 여기서는 색인어로 나타내고, 이런 특징이나 특성을 가진 문서의 통계를 보려고 할 때 사용합니다.

다음은 결제 수단에 대한 통계를 구하기 위한 예제입니다.

[Query DSL]

```
"aggs" : {
    "aggs_result" : {
        "terms" : {
            "field" : "payment_type",
            "order" : { "_count" : "desc" },
            "size" : 50
        }
    }
}
```

```
        }
    }
}
```

[Java API]

```
TermsBuilder aggsBuilder = AggregationBuilders.terms("aggs_result");
aggsBuilder.field("payment_type").size(50).order(Terms.Order.count(false));

String searchResult = Operators.executeAggregation(settings, client, new
MatchAllQueryBuilder(), aggsBuilder, "aggregations", "transactions");
```

실행 결과

```
"aggregations" : {
    "aggs_result" : {
        "doc_count_error_upper_bound" : -1,
        "sum_other_doc_count" : 0,
        "buckets" : [ {
            "key" : "card",
            "doc_count" : 6
        }, {
            "key" : "pay8",
            "doc_count" : 5
        }, {
            "key" : "bank",
            "doc_count" : 4
        }, {
            "key" : "cash",
            "doc_count" : 3
        }, {
            "key" : "phone",
            "doc_count" : 3
        } ]
    }
}
```

이 결과를 바탕으로 size 값에 따라 결과가 어떻게 달라지는지 알아보겠습니다.

제공된 예제에서 문서의 샤드⁰²별 색인어 정보는 다음 표와 같습니다.

표 2-2 샤드별 색인어 정보

구분	Shard 0	Shard 1	Shard 2
1	card : 3	card : 2	pay8 : 3
2	bank : 2	pay8 : 2	bank : 1
3	phone : 2	bank : 1	card : 1
4	cash : 1	cash : 1	cash : 1
5	-	phone : 1	-

Case 1. size=1

샤드별로 1번 행이 반환되므로 결과는 ‘card:5’가 되고, 전체 결과와 비교하면 card로 결제한 문서가 1개 적게 나옵니다.

Case 2. size=2

샤드별로 1번과 2번 행이 반환되므로 결과는 ‘card:5’, ‘pay8:5’, ‘bank:3’이 되고, 전체 결과와 비교하면 card와 bank가 1개씩 적게 나옵니다.

이 예제를 참고하여 결과가 왜 달라지는지 이해해보길 바랍니다.

경우의 수를 예로 설명한 것과 같이 aggregation의 크기는 최소한 샤드 크기와 같거나 크게 설정해야 합니다. 이를 작게 설정하면 샤드 크기로 강제 설정됩니다.

NOTE Terms Aggregation의 size

결과에 대한 얼마나 많은 term bucket을 출력할지 정의하는 것을 의미합니다. 이 크기에 따라 terms aggregation 결과가 달라지는데, 이는 고유한 색인어의 크기가 size 값보다 크면 term bucket 결과의 크기가 줄어들 수 있다는 것을 나타냅니다.

02 검색의 기본 데이터베이스가 되는 인덱스로, 큰 크기의 인덱스를 여러 개의 작은 인덱스로 나누어 저장하는 것을 의미한다.

2.1.7 Significant Terms Aggregation

검색 결과, 즉 문서에서 색인어에 대해 흥미롭거나 특이한 대상을 분석하는 용도로 사용하는 API입니다. 그러나 아직 실험적인 API이므로 사용할 때는 유의하길 바랍니다.

다음 예제는 두 가지 내용을 포함합니다.

- 상품명에 '자전거'가 있을 때 해당 상품의 카테고리 특이점 분석
- 전체 구매 상품 중 상품명에 대한 구매자의 성별 특성 분석

[Query DSL]

```
// case 1: 상품명에 "자전거"가 있을 때 해당 상품의 카테고리 특이점 분석
"aggs" : {
    "aggs_result" : {
        "significant_terms" : { "field" : "item_category" }
    }
}

// case 2: 전체 구매 상품 중 상품명에 대한 구매자의 성별 특성 분석
"aggs": {
    "aggs_result": {
        "terms": { "field": "item_name"},

        "aggs": {
            "sub_aggs_result": {
                "significant_terms": {"field": "buyer_gender"}
            }
        }
    }
}
```

[Java API]

```
// case 1: 상품명에 "자전거"가 있을 때 해당 상품의 카테고리 특이점 분석
TermQueryBuilder queryBuilder = QueryBuilders.termQuery("item_name", "자전거");
SignificantTermsBuilder aggsBuilder = AggregationBuilders.significantTerms("aggs_result");
aggsBuilder.field("item_category").size(50);

String searchResult = Operators.executeAggregation(settings, client, queryBuilder,
```

```

    aggsBuilder, "aggregations", "transactions");

// case 2: 전체 구매 상품 중 상품명에 대한 구매자의 성별 특성 분석
TermsBuilder termsBuilder = AggregationBuilders.terms("aggs_result");
termsBuilder.field("item_name");
SignificantTermsBuilder subAggsBuilder = AggregationBuilders.significantTerms("sub_aggs_result");
subAggsBuilder.field("buyer_gender");

termsBuilder.subAggregation(subAggsBuilder);

searchResult = Operators.executeAggregation(settings, client, new MatchAllQueryBuilder(),
termsBuilder, "aggregations", "transactions");

```

다음 결과가 나온 이유는 significant terms aggregation의 내부 처리 방법 때문인데, 색인어의 패턴을 분석하여 이런 결과가 나온다고만 이해하고 넘어가길 바랍니다.

실행 결과

```

// case 1: 상품명에 "자전거"가 있을 때 해당 상품의 카테고리 특이점 분석
"aggregations" : {
  "aggs_result" : {
    "doc_count" : 6,
    "buckets" : [ {
      "key" : "8A",
      "doc_count" : 4,
      "score" : 1.6666666666666665,
      "bg_count" : 4
    } ]
  }
}

// case 2: 전체 구매 상품 중 상품명에 대한 구매자의 성별 특성 분석
"aggregations" : {
  "aggs_result" : {
    "doc_count_error_upper_bound" : -1,
    "sum_other_doc_count" : 0,
    "buckets" : [ {
      "key" : "자전거",
      "doc_count" : 5,
      "sub_aggs_result" : {

```

```
"doc_count" : 5,
"buckets" : [ {
    "key" : "F",
    "doc_count" : 3,
    "score" : 0.3359999999999999,
    "bg_count" : 5
} ]
},
…중략…, {
    "key" : "안전모",
    …중략…
    "buckets" : [ {
        "key" : "F",
        …중략…
    }, {
        "key" : "매트",
        …중략…
        "buckets" : [ {
            "key" : "M",
            …중략…
        }
    }
]
```

2.1.8 Range Aggregation

범위 조건에 맞는 문서에 대한 통계 정보를 반환하는 API로 여러 개의 반환값을 갖습니다. 이 API는 숫자 필드에서만 사용할 수 있습니다.

다음은 결제 금액의 세 가지 범위 값에 대한 통계를 구하는 예제로, 각각의 범위는 `from`과 `to` 속성으로 지정할 수 있습니다.

[Query DSL]

```
"aggs" : {
    "aggs_result" : {
        "range" : {
            "field" : "payment_price",
            "ranges" : [
                { "from" : 0, "to" : 250000 },
                { "from" : 250001, "to" : 500000 },
                { "from" : 500001, "to" : 1000000 }
```

```
        ]
    }
}
}
```

주석 처리된 코드는 `unbounded`로 설정하여 지정된 값까지의 통계를 구할 수 있습니다.

[Java API]

```
RangeBuilder aggsBuilder = AggregationBuilders.range("aggs_result");
aggsBuilder.field("payment_price").addRange(0, 250000).addRange(250001, 500000).
addRange(500001, 1000000);
// aggsBuilder.field("payment_price").addUnboundedTo(250000).addUnboundedTo(500000).
addUnboundedTo(1000000);
```

`range` 설정 결과로는 범위에 속한 문서들에 대한 집계가 이루어지고, `unbounded` 설정 결과로는 범위에 속한 문서들의 누적 집계를 구합니다.

실행 결과

```
// range
"aggregations" : {
  "aggs_result" : {
    "buckets" : [ {
      "key" : "0.0-250000.0",
      "from" : 0.0,
      "from_as_string" : "0.0",
      "to" : 250000.0,
      "to_as_string" : "250000.0",
      "doc_count" : 19
    },
    ...
  }
}

...중략...

// unbounded
"aggregations" : {
  "aggs_result" : {
    "buckets" : [ {
      "key" : "*-250000.0",
      "to" : 250000.0,
```

```

    "to_as_string" : "250000.0",
    "doc_count" : 19
}, {
    "key" : "-500000.0",
    "to" : 500000.0,
    "to_as_string" : "500000.0",
    "doc_count" : 19
},
...중략...

```

2.1.9 Date Range Aggregation

이 aggregation은 날짜(date) 값을 가진 문서에 대한 통계 분석을 할 수 있습니다. 날짜 형식은 지정할 수 있는데, range aggregation과 달리 to 속성에 대한 값을 포함하지 않습니다. 날짜 형식과 패턴에 관한 자세한 내용은 [JodaDate⁰³](#)를 참고하기 바랍니다.

그림 2-1 JodaDate Table

Symbol	Meaning	Presentation	Examples
G	era	text	AD
C	century of era (>=0)	number	20
Y	year of era (>=0)	year	1996
x	weekyear	year	1996
w	week of weekyear	number	27
e	day of week	number	2
E	day of week	text	Tuesday; Tue
y	year	year	1996
D	day of year	number	189
M	month of year	month	July; Jul; 07
d	day of month	number	10
a	halfday of day	text	PM
K	hour of halfday (0-11)	number	0
h	clockhour of halfday (1-12)	number	12
H	hour of day (0-23)	number	0
k	clockhour of day (1-24)	number	24
m	minute of hour	number	30
s	second of minute	number	55
S	fraction of second	number	978
z	time zone	text	Pacific Standard Time; PST
Z	time zone offset/id	zone	-0800; -08:00; America/Los_Angeles
'	escape for text	delimiter	
''	single quote	literal	'

03 <http://bit.ly/1B0DpXr>

다음은 결제 시간을 기준으로 to 속성에 대한 값 제외(value exclusive) 기능과 2주 간격의 range aggregation 기능을 테스트하는 예제입니다.

format 속성은 날짜 값의 스타일을 설정합니다. keyed 속성은 range aggregation에도 적용되는데, 결과 출력 시 사용할 key명 사용을 설정합니다. key 속성은 출력 결과에 사용할 변수명을 설정합니다.

[Query DSL]

```
"aggs": {  
    "aggs_result": {  
        "date_range": {  
            "field": "payment_time",  
            "format": "yyyyMMddHHmmss",  
            "keyed": true,  
            "ranges": [  
                { "key": "target", "from": "20140604163000", "to": "20140604163001" },  
                { "key": "201406H1", "from": "20140531235959", "to": "20140616000000" },  
                { "key": "201406H2", "from": "20140615235959", "to": "20140701000000" }  
            ]  
        }  
    }  
}
```

[Java API]

```
DateRangeBuilder aggsBuilder = AggregationBuilders.dateRange("aggs_result");  
aggsBuilder.field("payment_time")  
    .format("yyyyMMddHHmmss")  
    .addRange("target", "20140604163000", "20140604163001")  
    .addRange("201406H1", "20140531235959", "20140616000000")  
    .addRange("201406H2", "20140615235959", "20140701000000");
```

설정한 key명으로 aggregation 결과가 반환된 것을 확인할 수 있습니다.

실행 결과

```
"aggregations" : {
```

```
"aggs_result" : {
    "buckets" : [ {
        "key" : "201406H1",
        "from" : 1.401580799E12,
        "from_as_string" : "20140531235959",
        "to" : 1.4028768E12,
        "to_as_string" : "20140616000000",
        "doc_count" : 15
    }, {
        "key" : "target",
        "from" : 1.4018994E12,
        "from_as_string" : "20140604163000",
        "to" : 1.401899401E12,
        "to_as_string" : "20140604163001",
        "doc_count" : 1
    }, {
        "key" : "201406H2",
        "from" : 1.402876799E12,
        "from_as_string" : "20140615235959",
        "to" : 1.4041728E12,
        "to_as_string" : "20140701000000",
        "doc_count" : 6
    } ]
}
}
```

2.1.10 Histogram Aggregation

여러 개의 반환값을 갖는 집계 API로, 문서에서 추출된 숫자 값에 적용할 수 있습니다. 이 aggregation은 interval 값에 따른 동적인 통계 정보를 반환합니다.

다음은 결제금액을 1만 원 단위의 구간별로 통계를 구하는 예제입니다.

[Query DSL]

```
"aggs" : {
    "aggs_result" : {
        "histogram" : {
            "field" : "payment_price",
```

```
        "interval" : 10000
    }
}
}
```

[Java API]

```
HistogramBuilder aggsBuilder = AggregationBuilders.histogram("aggs_result");
aggsBuilder.field("payment_price")
    .interval(10000);
```

histogram aggregation은 range aggregation에 대한 동적 구성을 할 수 있습니다. interval을 10000으로 주었으므로 key: 0의 범위는 0 ~ 9999, key: 10000의 범위는 10000 ~ 19999처럼 구간이 형성됩니다.

실행 결과

```
" aggregations" : {
    "aggs_result" : {
        "buckets" : [ {
            "key" : 0,
            "doc_count" : 2
        }, {
            "key" : 10000,
            "doc_count" : 1
        },
        ...
        ...
        {
            "key" : 190000,
            "doc_count" : 3
        }, {
            "key" : 510000,
            "doc_count" : 2
        }
    }
}
```

2.1.11 Date Histogram Aggregation

여러 개의 반환값을 갖는 집계 API입니다. histogram aggregation과 비슷한 기능을 수행하는데, histogram aggregation과 다르게 숫자 값이 아닌 날짜/시간(date/time) 값으로 interval 값을 설정하고, 이 값에 의한 동적 통계 정보를 반환하는 점이 다릅니다.

interval에는 year, quarter, month, week, day, hour, minute, second를 사용할 수 있습니다. 시간 단위(time unit)에 관한 자세한 내용은 [Elasticsearch 홈 페이지](#)⁰⁴를 참고하기 바랍니다.

시간대(time zone) 설정은 time_zone 파라미터를 이용하는데, 다음 두 가지로 설정할 수 있습니다.

- pre_zone : bucket 정보 획득 전에 시간대 설정
- post_zone : bucket 정보 획득 후에 시간대 설정

다음은 일 단위 결제 상품 수를 구하는 예제입니다.

[Query DSL]

```
"aggs" : {  
    "aggs_result" : {  
        "date_histogram" : {  
            "field" : "payment_time",  
            "interval" : "1d"  
        }  
    }  
}
```

[Java API]

```
DateHistogramBuilder aggsBuilder = AggregationBuilders.dateHistogram("aggs_result");  
aggsBuilder.field("payment_time")
```

04 <http://bit.ly/1BTg902>

```
.interval(Interval.DAY);
```

1일 단위로 분석한 결과를 확인할 수 있습니다.

실행 결과

```
"aggregations" : {  
    "aggs_result" : {  
        "buckets" : [ {  
            "key_as_string" : "20140601000000",  
            "key" : 1401580300000,  
            "doc_count" : 1  
        }, {  
            "key_as_string" : "20140602000000",  
            "key" : 1401667200000,  
            "doc_count" : 1  
        }, {  
            "key_as_string" : "20140603000000",  
            "key" : 1401753600000,  
            "doc_count" : 1  
        }  
        ...중략...  
    }  
}
```

2.1.12 Geo Distance Aggregation

여러 개의 반환값 갖는 집계 API로, range aggregation과 비슷한 기능을 수행하며 geo_point를 기준으로 거리에 대한 통계를 구합니다. date range aggregation과 동일하게 to 값은 범위에 포함하지 않습니다. 거리 단위의 기본 값은 m(meters)고, mi(miles), in(inch), yd(yards), km(kilometers), km(kilometers), cm(centimeters), mm(millimeters)도 사용할 수 있습니다.

distance_type은 sloppy_arc가 기본값이고, 그 외 arc, plane을 사용할 수 있습니다. arc는 정확도는 높지만 속도가 느리고, sloppy_arc는 arc보다 정확도는 떨어지지만 속도가 빠릅니다. plane은 가장 속도가 빠르지만 정확도가 떨어집니다.

다음은 지정한 지점을 기준으로 1000km 이내 구매 정보를 구하는 예제입니다.

[Query DSL]

```
"aggs" : {  
    "aggs_result" : {  
        "geo_distance" : {  
            "field" : "buyer_location.location",  
            "origin" : "35.907757,127.76692200000002",  
            "unit" : "km",  
            "distance_type" : "arc",  
            "ranges" : [  
                { "to" : 1000 }  
            ]  
        }  
    }  
}
```

[Java API]

```
GeoDistanceBuilder aggsBuilder = AggregationBuilders.geoDistance("aggs_result");  
aggsBuilder.field("buyer_location.location")  
.point("35.907757,127.76692200000002")  
.distanceType(GeoDistance.ARC)  
.unit(DistanceUnit.KILOMETERS)  
.addUnboundedTo(1000);
```

구매 관련 문서는 국가별로 작성되었으므로 1000km 이내로 범위를 넓게 구성하면 2개의 결과가 나오는 것을 확인할 수 있습니다.

실행 결과

```
" aggregations" : {  
    "aggs_result" : {  
        "buckets" : [ {  
            "key" : "->1000.0",  
            "from" : 0.0,  
            "to" : 1000.0,  
            "doc_count" : 2  
        } ]
```

```
    }  
}
```

2.2 Metric Aggregation

metric은 문서 내 숫자 필드의 값을 aggregation 유형에 따라 계산한 결과를 갖는 집계로, 이런 형식으로 통합하는 과정을 metric aggregation이라 합니다. metric aggregation에 대한 이해를 돋기 위해 간단히 stats aggregation 실행 결과를 살펴보겠습니다. 숫자 필드의 통계 정보가 계산된 결과를 확인할 수 있습니다.

stats aggregation 실행 결과

```
"aggregations" : {  
    "aggs_result" : {  
        "count" : 21,  
        "min" : 990.0,  
        "max" : 510250.0,  
        "avg" : 114580.47619047618,  
        "sum" : 2406190.0  
    }  
}
```

metric aggregation에는 다음과 같은 종류가 있는데, 이 중에서 일부를 살펴보겠습니다. 실행 가능한 전체 소스 코드는 제공된 코드를 참고하기 바랍니다.

Min, max, sum, avg, stats, extended stats, value count, percentile, percentile rank, cardinality, geo bounds, top hits, scripted metric

2.2.1 Min Aggregation

한 개의 반환값을 갖는 집계 API로, 집계 문서에서 추출된 수치 중 최소값을 반환

합니다. 즉, 질의 조건에 만족하는 문서의 최소값을 구할 때 유용한 API입니다.

다음은 전체 결제금액 중 최소금액을 구하는 예제입니다. 실행 결과 반환 시 이용할 변수명과 문서의 최소값을 구할 필드를 지정합니다.

[Query DSL]

```
"aggs" : {  
    "aggs_result" : { "min" : { "field" : "payment_price" } }  
}
```

[Java API]

```
Settings settings = Connector.buildSettings("elasticsearch");  
Client client = Connector.buildClient(settings, new String[] {"localhost:9300"});  
  
MinBuilder aggsBuilder = AggregationBuilders.min("aggs_result");  
aggsBuilder.field("payment_price");  
  
String searchResult = Operators.executeAggregation(settings, client, new  
MatchAllQueryBuilder(), aggsBuilder, "aggregations", "transactions");  
  
client.close();
```

지정한 변수명으로 결과가 반환된 것을 확인할 수 있습니다.

실행 결과

```
"aggregations" : {  
    "aggs_result" : {  
        "value" : 990.0  
    }  
}
```

2.2.2 Max Aggregation

한 개의 반환값을 갖는 집계 API로, 집계 문서에서 추출된 수치의 최대값을 반환합니다. 즉, 질의 조건에 만족하는 문서의 최대값을 구할 때 유용하며, aggregation 타입이 max라는 것 외에는 min aggregation과 동일합니다.

다음은 전체 결제금액 중 최대금액을 구하는 예제입니다.

[Query DSL]

```
"aggs" : {  
    "aggs_result" : { "max" : { "field" : "payment_price" } }  
}
```

[Java API]

```
...중략...  
MaxBuilder aggsBuilder = AggregationBuilders.max("aggs_result");  
aggsBuilder.field("payment_price");  
...중략...
```

실행 결과

```
"aggregations" : {  
    "aggs_result" : {  
        "value" : 510250.0  
    }  
}
```

2.2.3 Sum Aggregation

한 개의 반환값을 갖는 집계 API로, 집계 문서에서 추출된 수치의 합계를 반환합니다. 질의 조건에 만족하는 문서의 전체 합계를 구할 때 유용하며 aggregation 타입은 **sum**입니다.

다음은 전체 결제금액의 합계를 구하는 예제입니다.

[Query DSL]

```
"aggs" : {  
    "aggs_result" : { "sum" : { "field" : "payment_price" } }  
}
```

[Java API]

…중략…

```
SumBuilder aggsBuilder = AggregationBuilders.sum("aggs_result");
aggsBuilder.field("payment_price");
…중략…
```

실행 결과

```
" aggregations" : {
    "aggs_result" : {
        "value" : 2406190.0
    }
}
```

2.2.4 Avg Aggregation

한 개의 반환값을 갖는 집계 API로, 집계 문서에서 추출된 수치의 평균값을 반환합니다. 즉, 질의 조건에 만족하는 문서 전체에서 평균값을 구할 때 유용하며 aggregation 타입은 avg입니다.

다음은 전체 결제금액의 평균금액을 구하는 예제입니다.

[Query DSL]

```
"aggs" : {
    "aggs_result" : { "avg" : { "field" : "payment_price" } }
}
```

[Java API]

…중략…

```
AvgBuilder aggsBuilder = AggregationBuilders.avg("aggs_result");
aggsBuilder.field("payment_price");
…중략…
```

실행 결과

```
"aggregations" : {  
    "aggs_result" : {  
        "value" : 114580.47619047618  
    }  
}
```

2.2.5 Stats Aggregation

여러 개의 반환값을 갖는 집계 API로, 집계된 문서에서 추출된 수치의 통계 정보를 반환합니다. 질의 조건에 만족하는 문서의 전체 통계 정보를 구할 때 유용하고, 통계 정보는 count, min, max, sum, avg 타입의 속성으로 구성됩니다. aggregation 타입은 stats입니다.

다음은 전체 결제금액의 통계 정보를 구하는 예제입니다.

[Query DSL]

```
"aggs" : {  
    "aggs_result" : { "stats" : { "field" : "payment_price" } }  
}
```

[Java API]

```
...중략...  
StatsBuilder aggsBuilder = AggregationBuilders.stats("aggs_result");  
aggsBuilder.field("payment_price");  
...중략...
```

앞에서 소개한 여러 가지 단일 값(min, max, sum, avg) metric aggregation 타입을 하나의 API로 구한 것을 확인할 수 있습니다.

실행 결과

```
"aggregations" : {  
    "aggs_result" : {
```

```
        "count" : 21,  
        "min" : 990.0,  
        "max" : 510250.0,  
        "avg" : 114580.47619047618,  
        "sum" : 2406190.0  
    }  
}
```

2.2.6 Extended Stats Aggregation

여러 개의 반환값을 갖는 집계 API로, 집계된 문서에서 추출된 수치의 확장된 통계 정보를 반환합니다. 질의 조건을 만족하는 문서에서 통계 정보를 구할 때 유용하며, 통계 정보는 stats aggregation의 반환 속성과 sum_of_squares(제곱합), variance(분산), std_deviation(표준편차)로 구성됩니다. aggregation 타입은 extended_stats입니다.

다음은 전체 결제금액에 대한 통계 정보를 구하는 예제입니다.

[Query DSL]

```
"aggs" : {  
    "aggs_result" : { "extended_stats" : { "field" : "payment_price" } }  
}
```

[Java API]

```
…중략…  
ExtendedStatsBuilder aggsBuilder = AggregationBuilders.extendedStats("aggs_result");  
aggsBuilder.field("payment_price");  
…중략…
```

stats aggregation 결과와 비교하면 sum_of_squares, variance, std_deviation 값이 추가된 것을 확인할 수 있습니다.

실행 결과

```
"aggregations" : {  
    "aggs_result" : {  
        "count" : 21,  
        "min" : 990.0,  
        "max" : 510250.0,  
        "avg" : 114580.47619047618,  
        "sum" : 2406190.0,  
        "sum_of_squares" : 7.201483951E11,  
        "variance" : 2.1164095195011337E10,  
        "std_deviation" : 145478.84792990127  
    }  
}
```

2.2.7 Value Count Aggregation

한 개의 반환값을 갖는 집계 API로, 집계된 문서에서 추출된 수치값의 집계 결과를 반환합니다. 질의 조건에 만족하는 문서의 전체 평균값을 구해서 평균값을 구한 대상의 수를 얻을 때 적합한 API입니다. aggregation 타입은 value_count입니다.

이 aggregation은 집계 대상 필드 값이 ‘null’이면 집계 결과에는 포함되지 않습니다. 다음 예제에서 대상 문서의 구매국가(buyer_country)가 ‘null’일 때 value_count가 어떻게 집계되는지 확인해 보겠습니다. 이를 위해 문서 색인 시 한 문서에서 buyer_country 필드를 지정하지 않고 등록합니다. 여기서 사용한 buyer_country의 ‘null’ 특성은 이후 missing aggregation에서 활용합니다.

[Query DSL]

```
"aggs" : {  
    "aggs_result" : { "value_count" : { "field" : "buyer_country" } }  
}
```

…중략…

```
ValueCountBuilder aggsBuilder = AggregationBuilders.count("aggs_result");
aggsBuilder.field("buyer_country"); // 20개로 null 값이 하나 있음.
// aggsBuilder.field("payment_price"); // 21개
…중략…
```

'null' 값을 가진 필드가 있어서 결과는 20개로 나옵니다. 주석 처리한 'null' 값이 없는 payment_price로 테스트하면 결과는 21개가 나오게 됩니다. 이 예제를 avg aggregation과 함께 사용하면 평균을 구한 분모값을 확인할 수 있습니다.

실행 결과

```
"aggregations" : {
    "aggs_result" : {
        "value" : 20
    }
}
```

2.2.8 Percentiles Aggregation

여러 개의 반환값을 갖는 집계 API로, 집계된 문서에서 추출된 수치에 대한 하나 또는 여러 개의 백분율 통계 정보를 반환합니다. 질의 조건을 만족하는 문서에서 추출한 수치의 백분율 통계를 구할 때 유용하며 아직은 실험적인 API입니다. percentiles aggregation을 위한 필드는 반드시 숫자 타입이어야 합니다.

다음은 전체 결제금액을 백분율로 나누어 구간별 금액을 구하는 예제입니다.

```
"aggs" : {
    "aggs_result" : { "percentiles" : { "field" : "payment_price" } }
```

…중략…

```
PercentilesBuilder aggsBuilder = AggregationBuilders.percentiles("aggs_result");
aggsBuilder.field("payment_price");
…중략…
```

실행 결과

```
"aggregations" : {
    "aggs_result" : {
        "values" : {
            "1.0" : 1432.0,
            "5.0" : 3200.0,
            "25.0" : 26500.0,
            "50.0" : 39900.0,
            "75.0" : 175000.0,
            "95.0" : 510250.0,
            "99.0" : 510250.0
        }
    }
}
```

2.2.9 Cardinality Aggregation

한 개의 반환값을 갖는 집계 API로, 집계된 문서의 지정된 필드에서 구별되는 값의 통계 수치를 반환합니다. 질의 조건을 만족하는 문서 중 지정한 필드에서 중복 값을 제외하고 고유한 값만 집계할 때 적합하며, 아직은 실험적인 API입니다.

다음은 전체 주문 데이터 중에서 구매된 상품코드(item_code)의 종류가 몇 개인지 구매국가(buyer_country)는 몇 개국인지 알아보는 예제입니다.

```
"aggs" : {
    "aggs_result" : { "cardinality" : { "field" : "item_code" } }
```

```
//////  
"aggs" : {  
    "aggs_result" : { "cardinality" : { "field" : "buyer_country" } }  
}
```

[Java API]

…중략…

```
CardinalityBuilder aggsBuilder = AggregationBuilders.cardinality("aggs_result");
```

```
aggsBuilder.field("item_code");
```

…중략…

```
aggsBuilder = AggregationBuilders.cardinality("aggs_result");
```

```
aggsBuilder.field("buyer_country");
```

…중략…

실행 결과

```
// item_code에 대한 cardinality aggregation 결과
```

```
"aggregations" : {
```

```
    "aggs_result" : {
```

```
        "value" : 11
```

```
    }
```

```
}
```

```
// buyer_country에 대한 cardinality aggregation 결과
```

```
"aggregations" : {
```

```
    "aggs_result" : {
```

```
        "value" : 19
```

```
    }
```

```
}
```

2.3 정리

데이터의 규모가 커지면서 데이터 분석의 중요성이 매우 커지고 있습니다. 데이터는 가지고 있지만 이를 활용하지 못하는 경우가 매우 많습니다. 이런 이유로 SQL on Hadoop이 업계에서 화두가 되고 있는 게 아닌가 생각합니

다. Elasticsearch에서도 색인된 데이터에 대한 분석 기능을 제공하는데, 지금까지 살펴본 aggregations가 바로 그 기능입니다. 크게 두 가지 속성의 aggregation 방법을 제공하며 방법별로 다양한 API를 살펴보았습니다. Elasticsearch에서 제공하는 aggregations 기능을 활용하여 쉽고 간편하게 데이터 분석을 해보시길 바랍니다.

플러그인은 사용자 정의 방식으로 Elasticsearch의 기능을 강화하는 방법입니다. 플러그인은 만들어 사용할 수도 있고, 이미 정의된 모듈을 상속받아 기능적으로 변경할 수도 있어서 요구조건에 맞춰 Elasticsearch의 기능을 쉽게 변경하거나 추가할 수 있습니다. 물론 원본 기능을 그대로 사용해도 됩니다.

여기서는 Elasticsearch에서 가장 많이 사용하는 REST API 플러그인과 형태소 분석기 플러그인의 제작 방법을 알아보겠습니다.

3.1 Plugin 제작

플러그인을 만들기 위한 기본 프로젝트 구성을 살펴보겠습니다.

3.1.1 Plugin 프로젝트 생성

개발도구로 IDE는 이클립스 Eclipse, 프로젝트는 메이븐 Maven을 사용합니다. 개발도구의 환경 설정에 맞춰 프로젝트를 생성하고 테스트하면 됩니다.

기본적인 프로젝트 생성 흐름은 다음과 같습니다(실제 완성된 프로젝트는 제공된 [소스 코드](#)⁰¹에서 확인하기 바랍니다).

Step 1) 이클립스에서 메이븐 프로젝트를 생성합니다.

01 <http://bit.ly/1B52sbZ>

Step 2) 소스 폴더를 생성합니다.

- **src/main/resources/es-plugin.properties** 생성 : 제작할 플러그인 정보를 등록
- **src/main/assemblies/plugin.xml** 생성 : 다운로드 가능한 ZIP 파일 형태로 등록

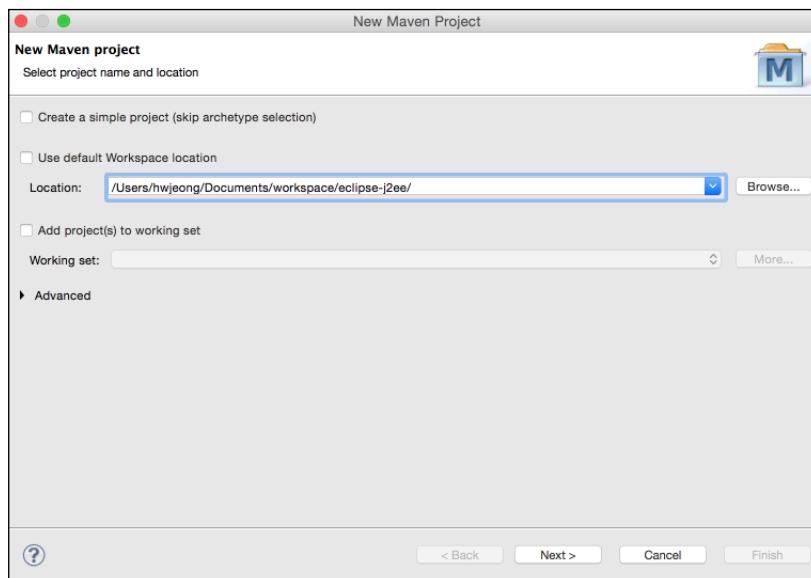
Step 3) 플러그인 등록을 위해 AbstractPlugin을 상속받은 plugin 클래스를 생성합니다.

Step 4) 구현하려는 기능의 플러그인을 작성한다.

3.1.2 Plugin 프로젝트 구성

플러그인 구현을 위한 메이븐 프로젝트를 구성합니다.

그림 3-1 메이븐 프로젝트 생성



Archetype은 ‘Maven-archetype-quickstart’를 선택합니다.

그림 3-2 Archetype 선택

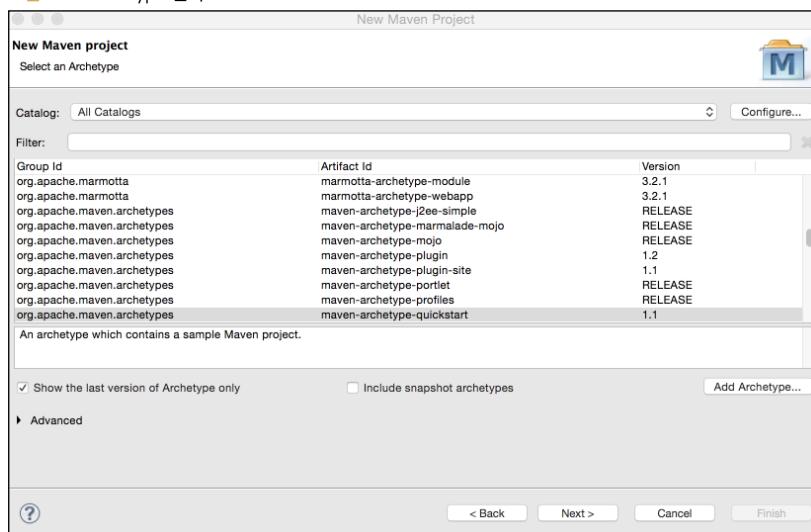


그림 3-3 Archetype 파라미터 등록

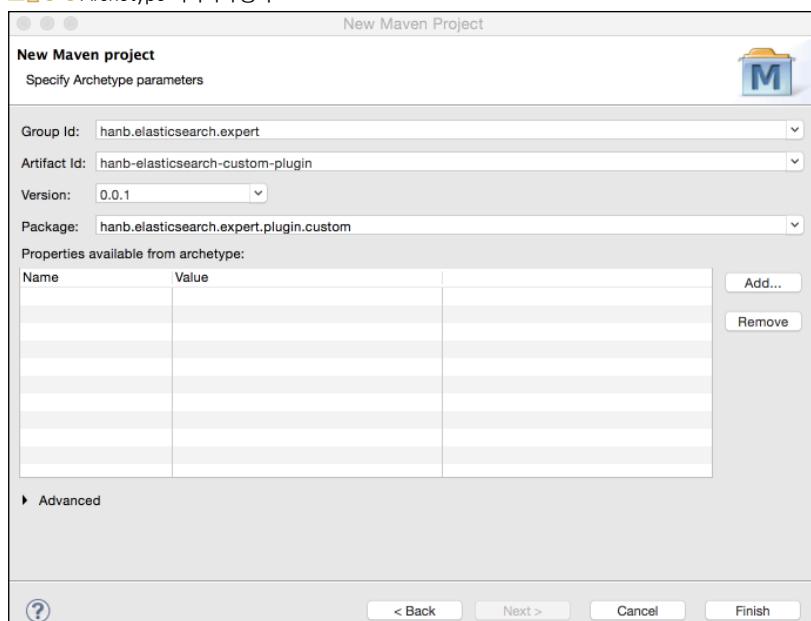
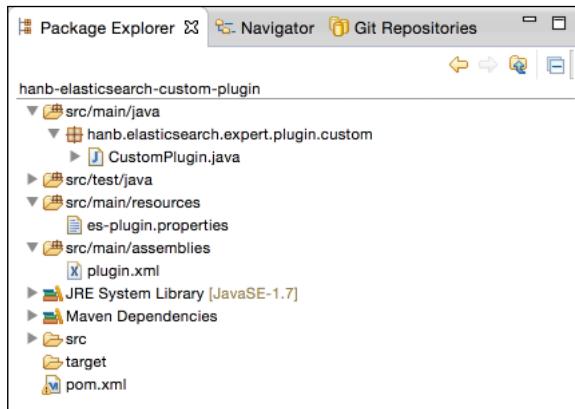


그림 3-4 Maven 프로젝트 생성 후 기본 구성이 끝난 화면



메이븐 프로젝트 생성이 끝나면 es-plugin.properties과 plugin.xml 파일을 별도로 생성합니다. 플러그인 등록을 위한 CustomPlugin.java 파일은 개발 요구사항에 맞춰 작성하고, pom.xml 파일은 개발 환경에 맞춰 의존성(dependency)을 등록합니다. 플러그인 템플릿(plugin template)은 프로젝트를 생성하면서 직접 만들거나 제공된 소스 코드를 받아서 사용할 수 있습니다.

3.2 REST Plugin 만들기

Elasticsearch의 가장 좋은 장점은 REST API입니다. REST API를 이용해 다양한 클라이언트 프로그램에서 Elasticsearch를 사용할 수 있습니다. 여기서는 클라이언트 프로그램에서 사용할 수 있는 REST 플러그인을 만들어 보고 이를 Elasticsearch에 적용하는 방법을 알아보겠습니다.

3.2.1 REST Plugin 프로젝트 생성과 등록

Custom 플러그인 템플릿을 이용하여 만들어 보겠습니다. 여기서 만드는 플러그인은 HTTP 요청을 받아 데몬이 정상적으로 응답하는지 검사하는 Health Check용 REST API입니다.

Health Check용 플러그인을 등록하기 위한 클래스를 생성합니다. Abstract Plugin을 상속받으면 기본으로 다음 두 메서드를 구현해야 합니다.

- **String description()** : 플러그인의 설명을 작성합니다.
- **String name()** : 플러그인의 이름을 작성합니다.

모듈을 등록하기 위해 추가로 다음 두 메서드를 작성해야 합니다.

- **void processModule(Module module)** : 플러그인 인터페이스에 선언되어 있으며, AbstractPlugin을 상속받아 오버라이드합니다.
- **void onModule(RestModule module)** : PluginService를 통해 등록합니다.

[HealthCheckPlugin.java]

```
public class HealthCheckPlugin extends AbstractPlugin {  
    @Override  
    public String name() {  
        return "health-check";  
    }  
  
    @Override  
    public String description() {  
        return "elasticsearch(http) health check.";  
    }  
  
    // 등록하는 방식은 다음 두 가지 모두 동작함.  
    @Override  
    public void processModule(Module module) {  
        if (module instanceof RestModule) {  
            ((RestModule) module).addRestAction(HealthCheckAction.class);  
        }  
    }  
  
    // public void onModule(RestModule module) {  
    //     module.addRestAction(HealthCheckAction.class);  
    // }  
}
```

3.2.2 REST Plugin 기능 구현

BaseRestHandler를 상속받아 HealthCheckAction을 구현해 보겠습니다.

BaseRestHandler를 상속받으면 기본으로 다음 메서드를 구현해야 합니다.

- **void handleRequest(RestRequest …, RestChannel …, Client …)** : 요청을 처리하고 결과를 반환합니다.

이 REST API를 이용하려면 RestController의 registerHandler() 메서드를 통해 클라이언트에서 요청할 URI를 등록해야 하는데, GET 방식의 '/_healthcheck'로 등록합니다.

[HealthCheckAction.java]

```
public class HealthCheckAction extends BaseRestHandler {  
    @Inject  
    protected HealthCheckAction(Settings settings, Client client, RestController controller) {  
        super(settings, client);  
        controller.registerHandler(RestRequest.Method.GET, "/_healthcheck", this);  
    }  
  
    @Override  
    public void handleRequest(RestRequest request, RestChannel channel, Client client) {  
        channel.sendResponse(new BytesRestResponse(RestStatus.OK, "OK"));  
        return;  
    }  
}
```

3.2.3 REST Plugin 등록 설정

Elasticsearch에서 별도의 등록 설정 없이 플러그인이 자동으로 로딩될 수 있도록 properties 정보를 작성하고 bin/plugin으로 설치되도록 plugin.xml을 설정합니다.

```
[ src/main/resources/es-plugin.properties ] ——————
```

```
plugin=hanb.elasticsearch.expert.plugin.registry.HealthCheckPlugin
```

```
[ src/main/assemblies/plugin.xml ] ——————
```

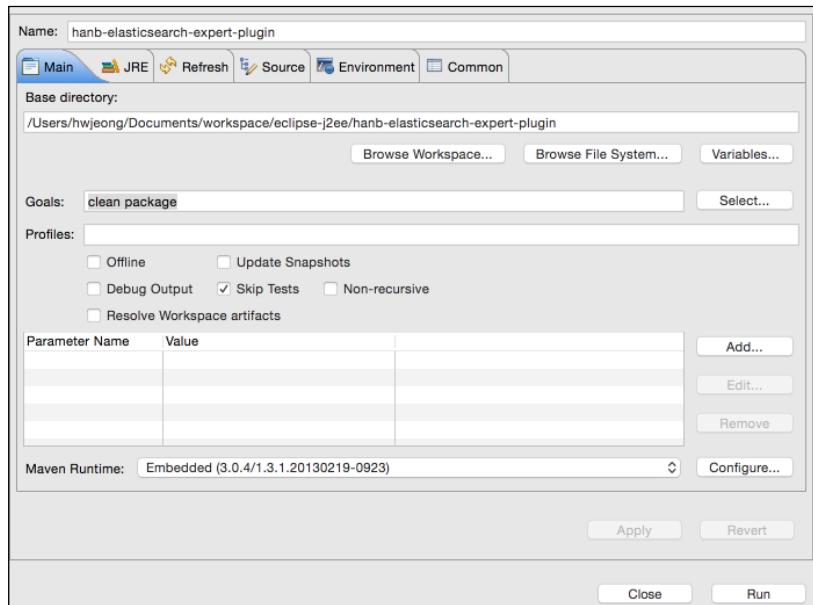
```
<?xml version="1.0"?>
<assembly>
  <id>plugin</id>
  <formats>
    <format>zip</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
  <dependencySets>
    <dependencySet>
      <outputDirectory>/</outputDirectory>
      <useProjectArtifact>true</useProjectArtifact>
      <useTransitiveFiltering>true</useTransitiveFiltering>
      <excludes>
        <exclude>org.elasticsearch:elasticsearch</exclude>
      </excludes>
    </dependencySet>
    <dependencySet>
      <outputDirectory>/</outputDirectory>
      <useProjectArtifact>true</useProjectArtifact>
      <scope>provided</scope>
    </dependencySet>
  </dependencySets>
</assembly>
```

3.2.4 REST Plugin 빌드와 설치

기능 구현을 완료했으니 빌드 후 설치해 보겠습니다.

먼저 [hanb-elasticsearch-expert-plugin → Run as → Maven build]를 선택하여 빌드합니다.

그림 3-5 메이븐 빌드 설정



설치는 다음 단계로 진행합니다.

Step 1) Elasticsearch가 설치된 곳의 plugins 폴더 아래 health-check라는 폴더를 만듭니다.

health-check 폴더 생성

```
elasticsearch$ mkdir -p plugins/health-check
```

Step 2) 플러그인의 빌드 결과물인 hanb-elasticsearch-expert-plugin-0.0.1.jar 파일을 plugins/health-check 폴더로 복사합니다.

Step 3) Elasticsearch를 재시작한 후 다음 요청으로 동작 상태를 확인합니다.

동작 확인

```
$ curl -XGET http://localhost:9200/_healthcheck
```

정상적으로 동작하면 결과는 'OK'를 반환합니다.

3.2.5 REST Plugin 구현 요약

Package

- `hanb.elasticsearch.expert.plugin.registry` : plugin 등록을 위한 패키지입니다.
- `hanb.elasticsearch.expert.rest` : rest action 생성을 위한 패키지입니다.

Class

- `HealthCheckPlugin` : elasticsearch에 custom rest action을 등록하기 위한 클래스입니다.
- `HealthCheckAction` : custom rest action 기능 구현을 위한 클래스입니다.

Resources

- `es-plugin.properties` : elasticsearch 실행 시 플러그인을 등록하기 위한 정보입니다.

Assembly

- `plugin.xml` : 메이븐 빌드 시 bin/plugin으로 설치되도록 ZIP 파일을 생성하기 위한 정보입니다.

Installation

- **플러그인 폴더 생성** : plugins/health-check로 폴더를 생성합니다. `HealthCheckPlugin` 클래스에서 `name()`에 해당하는 정보입니다.

설치

빌드 결과물인 JAR 파일을 해당 플러그인 폴더로 복사하거나 ZIP 파일을 웹 서버에 등록한 후 다음 명령으로 설치합니다.

설치 명령

```
bin/plugin --install http://localhost:8080/hanb-elasticsearch-expert-plugin-0.0.1.zip
```

Elasticsearch를 재시작하고 다음 명령으로 정상적으로 동작하는지 확인합니다.

동작 확인

```
$ curl -XGET http://localhost:9200/_healthcheck
```

3.3 Analyzer Plugin 만들기

루씬에서는 매우 많은 종류의 유용한 analyzer를 제공합니다. 가장 일반적인 analyzer로는 StandardAnalyzer가 있고, 한글분석기로는 전통적으로 CJKAnalyzer를 이용하거나 이수명 님이 오픈소스로 제공하는 한글 형태소 분석기가 있습니다. 현재 루씬 프로젝트로 정식 컨트리뷰트^{Contribute}가 되어서 ‘Arirang’이라는 이름으로 제공됩니다. 여기서는 JAR 파일 형태로 제공된 한글 형태소 분석기를 이용해 elasticsearch-analysis-arirang이라는 플러그인을 만들어보겠습니다.

NOTE Arirang Analyzer

공식 카페: <http://cafe.naver.com/korlucene>

Repository: <https://lucenekorean.svn.sourceforge.net/svnroot/lucenekorean/>

3.3.1 Analyzer Plugin 프로젝트 생성과 등록

Custom 플러그인 템플릿을 이용하여 만들어보겠습니다. 이 플러그인은 Arirang analyzer를 그대로 Elasticsearch analyzer 플러그인으로 래핑^{wrapping}해서 만드는 한글 형태소 분석기입니다(전체 코드는 제공된 [소스 코드](#)⁰²를 참고하기 바랍니다).

먼저 analyzer 플러그인을 등록하기 위한 클래스를 생성합니다. Abstract

02 <http://bit.ly/1DXLpZf>

Plugin을 상속받으면 기본으로 다음 두 메서드를 구현해야 합니다.

- **String description()**: 플러그인의 설명을 작성합니다.
- **String name()**: 플러그인의 이름을 작성합니다.

이 두 메서드만으로 모듈을 등록할 수 없으므로 추가 메서드를 작성해야 합니다.

REST 플러그인에서 `processModule(...)` 메서드를 사용했으므로 여기서는 `onModule(...)` 메서드를 사용해 보겠습니다.

[AnalysisArirangPlugin.java]

```
public class AnalysisArirangPlugin extends AbstractPlugin {  
  
    @Override  
    public String name() {  
        return "analysis-arirang";  
    }  
  
    @Override  
    public String description() {  
        return "Korean Analyzer";  
    }  
  
    public void onModule(AnalysisModule module) {  
        module.addProcessor(new ArirangAnalysisBinderProcessor());  
    }  
}
```

3.3.2 Analyzer Plugin 기능 구현

보통 루씬에서 analyzer 구성은 다음과 같습니다.

- **Analyzer** : 하나의 tokenizer와 여러 개의 토큰 필터로 구성되며, 문자열을 분석하고 색인어를 추출합니다.
- **Tokenizer** : Reader를 통해서 입력된 토큰 스트림으로 입력된 문자열에 대한 분석 작업과 분리 작업을 하여 색인어를 만듭니다.
- **TokenFilter** : Tokenizer에 의해 분리된 토큰 스트림을 정의한 토큰 필터를 적용하여 최종 색인어를 만듭니다.

이처럼 하나의 analyzer는 tokenizer와 토큰 필터를 이용해서 구성하는데, 한글 형태소 분석기 플러그인 역시 Arirang의 analyzer, tokenizer, 토큰 필터를 이용해서 구현합니다.

analyzer 플러그인은 다음 4개의 클래스 생성해야 합니다.

- **AnalysisBinderProcessor** : Elasticsearch로 analyzer를 등록하는 역할을 합니다.
- **AnalyzerProvider** : Arirang analyzer를 생성하는 역할을 합니다.
- **TokenizerFactory** : Arirang tokenizer를 생성하는 역할을 합니다.
- **TokenFilterFactory** : Arirang 토큰 필터를 생성하는 역할을 합니다.

먼저 analyzer provider를 구현해 보겠습니다.

[ArirangAnalyzerProvider.java]

```
public class ArirangAnalyzerProvider extends AbstractIndexAnalyzerProvider<KoreanAnalyzer> {  
  
    private final KoreanAnalyzer analyzer;  
  
    @Inject  
    public ArirangAnalyzerProvider(Index index, @IndexSettings Settings indexSettings,  
        Environment env, @Assisted String name, @Assisted Settings settings) throws IOException {  
        super(index, indexSettings, name, settings);  
  
        analyzer = new KoreanAnalyzer(Lucene.VERSION.LUCENE_4_9);  
    }  
  
    @Override  
    public KoreanAnalyzer get() {  
        return this.analyzer;  
    }  
}
```

내부적으로는 루씬의 analyzer를 상속받으며, 다중 상속 형태로 Korean Analyzer를 상속받아 ArirangAnalyzerProvider를 구현해야 합니다. 다음은 기본으로 구현해야 하는 메서드입니다.

- **KoreanAnalyzer get()** : KoreanAnalyzer의 analyzer 객체를 얻습니다.

다음은 tokenizer factory를 구현해 보겠습니다. ArirangAnalyzer에서 사용할 tokenizer를 등록하려면 ArirangTokenizerFactory를 생성하고, tokenizer로 KoreanTokenizer를 생성하여 사용합니다. 다음은 tokenizer 생성을 위해 기본으로 구현해야 하는 메서드입니다.

- **Tokenizer create(Reader ...)** : KoreanTokenizer 객체를 생성합니다.

[ArirangTokenizerFactory.java]

```
public class ArirangTokenizerFactory extends AbstractTokenizerFactory {  
  
    @Inject  
    public ArirangTokenizerFactory(Index index, @IndexSettings Settings indexSettings, @  
Assisted String name, @Assisted Settings settings) {  
        super(index, indexSettings, name, settings);  
    }  
  
    @Override  
    public Tokenizer create(Reader reader) {  
        return new KoreanTokenizer(reader);  
    }  
}
```

세 번째로 토큰 필터를 구현합니다. ArirangTokenizer에서 사용할 토큰 필터를 등록하려면 ArirangTokenFilterFactory를 생성하고, 토큰 필터로 KoreanFilter를 생성합니다. 다음은 토큰 필터를 생성하는 데 기본으로 구현해야 하는 메서드입니다.

- **TokenStream create(TokenStream ...)** : KoreanFilter 객체를 생성합니다.

[ArirangTokenFilterFactory.java]

```
public class ArirangTokenFilterFactory extends AbstractTokenFilterFactory {  
  
    @Inject  
    public ArirangTokenFilterFactory(Index index, @IndexSettings Settings indexSettings,  
@Assisted String name, @Assisted Settings settings) {  
        super(index, indexSettings, name, settings);  
    }  
}
```

```
    @Override
    public TokenStream create(TokenStream tokenStream) {
        return new KoreanFilter(tokenStream);
    }
}
```

여기까지 Arirang analyzer의 세 가지 주요 클래스를 구성하였습니다. 이제 이 클래스들을 Elasticsearch에 바인딩^{binding}하기 위한 등록 클래스를 작성합니다.

[ArirangAnalysisBinderProcessor.java]

```
public class ArirangAnalysisBinderProcessor extends AnalysisModule, AnalysisBinderProcessor {
    @Override
    public void processAnalyzers(AnalyzersBindings analyzersBindings) {
        analyzersBindings.processAnalyzer("arirang_analyzer", ArirangAnalyzerProvider.
class);
    }
    @Override
    public void processTokenizers(TokenizersBindings tokenizersBindings) {
        tokenizersBindings.processTokenizer("arirang_tokenizer", ArirangTokenizerFactory.
class);
    }
    @Override
    public void processTokenFilters(TokenFiltersBindings tokenFiltersBindings) {
        tokenFiltersBindings.processTokenFilter("arirang_filter",
ArirangTokenFilterFactory.class);
    }
}
```

analyzer, tokenizer, 토큰 필터를 Elasticsearch에서 사용하려면 앞에서 구현한 각각의 binder를 통해 Arirang analyzer를 등록하게 됩니다.

3.3.3 Analyzer Plugin 등록 설정

Elasticsearch에서 별도의 등록 설정 없이 자동으로 플러그인이 로딩될 수 있도록 properties 정보를 작성하고 bin/plugin으로 설치되도록 plugin.xml을 설정합니다.

[src/main/resources/es-plugin.properties]

plugin=org.elasticsearch.plugin.analysis.arirang.AnalysisArirangPlugin

배포 패키지를 생성하려면 Arirang analyzer 관련 dependency jar 파일을 함께 포함해야 하고, 그 외 불필요한 dependency jar 파일은 제외해야 합니다. 다음의 assembly 플러그인에서는 두 개의 arirang dependency jar 파일과 한 개의 analyzer plugin jar 파일이 배포 패키지로 생성됩니다.

[src/main/assemblies/plugin.xml]

```
<?xml version="1.0"?>
<assembly>
    <id>plugin</id>
    <formats>
        <format>zip</format>
    </formats>
    <includeBaseDirectory>false</includeBaseDirectory>
    <fileSets>
        <fileSet>
            <directory>lib</directory>
            <outputDirectory>/</outputDirectory>
            <includes>
                <include>arirang_lucene-analyzer-4.9.0.jar</include>
                <include>morph-1.0.0.jar</include>
            </includes>
        </fileSet>
    </fileSets>
    <dependencySets>
        <dependencySet>
            <outputDirectory>/</outputDirectory>
            <useProjectArtifact>true</useProjectArtifact>
            <useTransitiveFiltering>true</useTransitiveFiltering>
            <excludes>
                <exclude>org.elasticsearch:elasticsearch</exclude>
            </excludes>
        </dependencySet>
        <dependencySet>
            <outputDirectory>/</outputDirectory>
            <useProjectArtifact>true</useProjectArtifact>
```

```
<useTransitiveFiltering>true</useTransitiveFiltering>
<includes>
    <include>org.apache.lucene:lucene-analyzers-arirang</include>
</includes>
</dependencySet>
</dependencySets>
</assembly>
```

3.3.4. Analyzer Plugin 빌드와 설치

기능 구현을 완료하였으니 빌드한 후 설치해 보겠습니다.

플러그인을 설치하기 전에 다음 명령을 통해 작성한 플러그인을 빌드합니다. 빌드 옵션은 Maven goals : clean {package or install}입니다.

빌드 명령

```
$ mvn clean install -DskipTests
```

명령 수행이 끝나면 [elasticsearch-analysis-arirang → Run as → Maven build]를 선택하여 빌드하고 다음 단계로 설치를 진행합니다.

Step 1) Elasticsearch가 설치된 곳의 plugins 폴더 아래 analysis-arirang이라는 폴더를 만듭니다.

폴더 생성

```
elasticsearch$ mkdir -p plugins/analysis-arirang
```

Step 2) assembly 플러그인의 빌드 결과물인 elasticsearch-analysis-arirang-1.0.0.zip 파일을 plugins/analysis-arirang 폴더로 복사하여 압축을 풉니다.

압축 해제

```
analysis-arirang$ cp elasticsearch-analysis-arirang/target/releases/elasticsearch-
analysis-arirang-1.0.0.zip .
analysis-arirang$ unzip elasticsearch-analysis-arirang-1.0.0.zip
Archive:  elasticsearch-analysis-arirang-1.0.0.zip
  inflating: elasticsearch-analysis-arirang-1.0.0.jar
  inflating: arirang_lucene_analyzer-4.9.0.jar
  inflating: morph-1.0.0.jar
analysis-arirang$ ls
arirang_lucene_analyzer-4.9.0.jar
elasticsearch-analysis-arirang-1.0.0.jar
elasticsearch-analysis-arirang-1.0.0.zip
morph-1.0.0.jar
```

Step 3) Arirang analyzer를 테스트하기 위한 test 인덱스를 생성합니다.

인덱스 생성

```
curl -XPUT http://localhost:9200/test -d '{
  "settings" : {
    "index": {
      "analysis": {
        "analyzer": {
          "arirang_analyzer": {
            "type": "arirang_analyzer",
            "tokenizer": "arirang_tokenizer",
            "filter": ["trim", "lowercase", "arirang_filter"]
          }
        }
      }
    }
  }
}'
```

기능을 점검하기 위해 test 인덱스를 생성하였으나 실제로 운영하려면 test 인덱스 대신 실제로 색인할 대상 인덱스를 지정하여 등록합니다.

Step 4) Elasticsearch를 재시작하고 다음 요청으로 동작 상태를 확인합니다.

```
$ curl -XGET 'http://localhost:9200/test/\_analyze?analyzer=arirang-analyzer&text=%EB%8B%A4'  
arirang analyzer 플러그인 테스트입니다. '
```

실행 결과는 다음과 같습니다.

그림 3-6 Analyzer 플러그인 설치 결과

```
{
  "tokens" : [ {
    "token" : "루",
    "start_offset" : 0,
    "end_offset" : 2,
    "type" : "<KOREAN>",
    "position" : 1
  }, {
    "token" : "어",
    "start_offset" : 3,
    "end_offset" : 10,
    "type" : "<ALPHANUM>",
    "position" : 2
  }, {
    "token" : "리",
    "start_offset" : 11,
    "end_offset" : 19,
    "type" : "<ALPHANUM>",
    "position" : 3
  }, {
    "token" : "ang",
    "start_offset" : 20,
    "end_offset" : 23,
    "type" : "<KOREAN>",
    "position" : 4
  }, {
    "token" : " ",
    "start_offset" : 25,
    "end_offset" : 28,
    "type" : "<KOREAN>",
    "position" : 5
  }, {
    "token" : "테",
    "start_offset" : 29,
    "end_offset" : 30,
    "type" : "<KOREAN>",
    "position" : 6
  }, {
    "token" : "스",
    "start_offset" : 29,
    "end_offset" : 30,
    "type" : "<KOREAN>",
    "position" : 6
  } ] }
```

3.4 정리

플러그인 적용은 Elasticsearch의 매우 유용하고 강력한 기능 중 하나입니다. 이 장에서 설명한 REST 플러그인과 analyzer 플러그인 구현 방법을 참고하여 다양한 플러그인을 구현할 수 있으므로 필요한 기능이 있을 때 활용해 보길 바랍니다.

NOTE **오픈소스 기반 한글 형태소 분석기**

앞에 소개한 Arirang 외에도 여러 한글 형태소 분석기가 있으니 참고하길 바랍니다.

twitter korean text : <https://github.com/twitter/twitter-korean-text>

komoran : <https://github.com/shineware/komoran-2.0>

Hadoop 연동

Elasticsearch에서는 HDFS^{Hadoop Distributed File System}에 저장된 데이터를 다루기 위해 Elasticsearch 하둡 플러그인을 제공합니다. 이 플러그인은 Elasticsearch를 다양한 하둡 소프트웨어와 통합할 수 있게 도와줍니다.

NOTE Elasticsearch에서 지원하는 하둡 소프트웨어

다음 소프트웨어는 안정 버전과 베타 버전에서 모두 제공합니다.

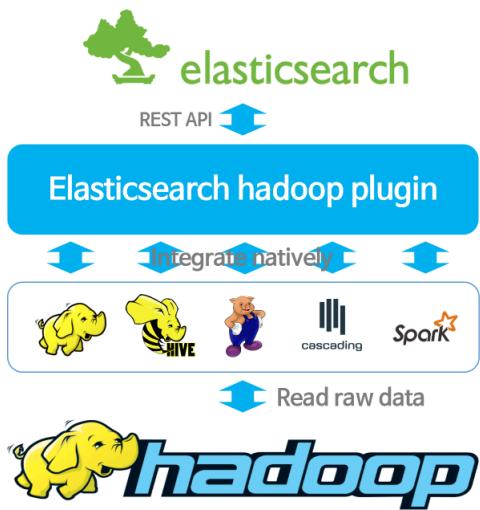
- MapReduce
- Hive
- Pig
- Cascading
- Spark (elasticsearch-hadoop-2.1.0.Beta1)
- Storm (elasticsearch-hadoop-2.1.0.Beta2)

색인

기본적으로 Elasticsearch는 하둡으로 직접 색인할 수 없습니다. Elasticsearch는 루씬 라이브러리에 의한 인덱스 파일 구조를 사용하므로 색인 시 random access write가 가능해야 합니다. 하지만 하둡은 random access read만 지원하고 write는 지원하지 않아서 중간에 하둡 소프트웨어를 이용하여 Elasticsearch로 색인할 수 있게 플러그인을 제공하고 있습니다.

기본 과정은 HDFS에서 데이터를 읽어서 Elasticsearch로 bulk 요청을 보내 색인합니다. 다음 그림은 하둡에 있는 데이터를 읽어서 Elasticsearch로 색인하는 과정은 보여줍니다.

그림 4-1 Hadoop 색인 연동 아키텍처



검색 질의

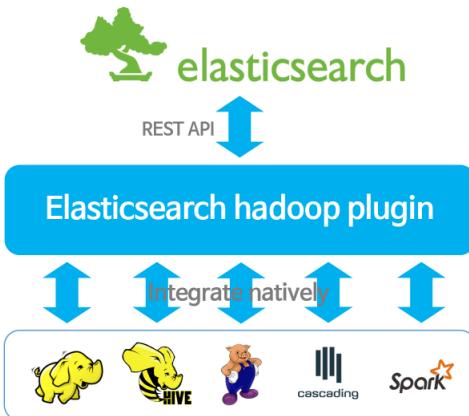
검색 질의는 하둡이 아닌 Elasticsearch에서 색인이 완료된 데이터를 대상으로 수행합니다. 검색 질의 역시 하둡 소프트웨어 라이브러리와 Elasticsearch 하둡 플러그인을 통해서 질의 요청을 합니다.

NOTE 연동 가능한 하둡 벤더와 버전

- Apache Hadoop 2.4.x, 2.2.x, 1.2.x, 1.1.x
- Amazon EMR 3.1.x, 3.0.x, 2.4.x
- Cloudera CDH 5.1.x, 5.0.x, 4.5.x, 4.4.x, 4.2.2
- Hortonworks HDP 2.1.x, 2.0.x, 1.3.x
- Greenplum GPHD 1.2
- Intel Hadoop 2.5.1
- Pivotal HD 2.1.x, 2.0.x, 1.1.x
- MapR 4.0.x, 3.1.x, 3.0.x, 2.1.x

검색 질의 과정은 Elasticsearch 검색 요청과 비슷하고, 단지 질의를 하둡 소프트웨어를 이용하여 Elasticsearch로 합니다. 다음 그림은 이 과정을 나타냅니다.

그림 4-2 Hadoop 질의 연동 아키텍처



Elasticsearch와 하둡을 연동하는 기본 개요를 간단히 살펴보았습니다. 이를 참고하여 하둡 소프트웨어를 이용해 색인과 검색 질의를 구현하고, 쉽게 다뤄볼 수 있는 맵리듀스^{MapReduce}와 Hive를 이용한 연동 방법을 알아보겠습니다. 전체 소스 코드는 제공된 [소스 코드](#)⁰¹를 참고하기 바랍니다.

4.1 MapReduce 연동

맵리듀스는 HDFS에 저장된 데이터를 처리하는 가장 대표적인 프레임워크로, 맵리듀스를 사용하려면 하둡이 설치되어 있어야 합니다. 하둡 설치 방법은 아파치 하둡 공식 문서인 [single node setup](#)⁰²을 참고하길 바라며 여기서는 설치 관련 내용은 다루지 않습니다.

01 <http://bit.ly/1zHvFob>

02 <http://bit.ly/1G0fGXf>

4.1.1 준비 항목

맵리듀스 구현을 위해서는 기본으로 다음 항목을 준비해야 합니다.

- **Hadoop 1.x 또는 2.x 설치** : 테스트에는 1.2.1과 2.3.0이 사용됩니다. 설치 및 준비가 어려우면 로컬 맵리듀스 테스트로 진행합니다.
- **Elasticsearch hadoop plugin 2.0.2 dependency 등록** : 테스트에는 2.0.0과 2.0.2 모두 정상적으로 사용할 수 있습니다.
- **JDK 1.7.0_25 이상**
- **Elasticsearch 0.90.x 이상** : 1.3.x 이상의 최신 버전 사용을 추천합니다.

4.1.2 색인 MapReduce 구현

색인용 맵리듀스를 작성하기 위해 `IndexWriter`라는 메인 메서드를 가지는 클래스를 작성합니다. `IndexWriter`는 색인 작업만 수행하므로 별도의 reducer 구현은 필요 없고, mapper만 구현합니다.

다음 코드는 Elasticsearch 하둡 플러그인의 맵리듀스 통합 테스트 코드와 같으며 독립적으로 실행할 수 있게 일부 코드를 수정하였습니다. 하둡을 설치할 필요 없이 이클립스에서 로컬 맵리듀스로 동작합니다.

[`IndexWriter.java` – 메인 메서드]

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    conf.setBoolean("mapred.map.tasks.speculative.execution", false);
    conf.setBoolean("mapred.reduce.tasks.speculative.execution", false);
    conf.set(ConfigurationOptions.ES_NODES, "localhost:9200");
    conf.set(ConfigurationOptions.ES_RESOURCE, "radio/artists");
    HadoopCfgUtils.setGenericOptions(conf);

    Job job = new Job(conf);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(EsOutputFormat.class);
    job.setMapOutputValueClass(LinkedMapWritable.class);
    job.setMapperClass(TabMapper.class);
    job.setNumReduceTasks(0);
```

```

// job split
File f1 = new File("data/artists.dat");
long splitSize = f1.length() / 3;
TextInputFormat.setMaxInputSplitSize(job, splitSize);
TextInputFormat.setMinInputSplitSize(job, 50);

// text
Job standard = new Job(job.getConfiguration());
standard.setMapperClass(TabMapper.class);
standard.setMapOutputValueClass(LinkedMapWritable.class);
TextInputFormat.addInputPath(standard, new Path("data/artists.dat"));

boolean result = standard.waitForCompletion(true);
}

```

TextInputFormat의 InputSplitSize는 job 실행 시 task 수를 조정하기 위해 사용합니다. snappy 등의 압축 파일로 되어 있을 경우 의미가 없으며 단순 텍스트 파일일 때만 동작하므로 주의합니다. 또한, 로컬 맵리듀스에서도 1개의 map task만 수행합니다.

다음은 텍스트 파일에 탭 구분자가 있는 문서를 맵리듀스로 색인하는 프로그램을 작성한 예제입니다.

radio/artists

- radio는 인덱스고, artists는 타입이 됩니다.
- 사전 매핑 정의를 하지 않고 동적 매핑⁰³Dynamic Mapping⁰³을 이용해서 구성합니다.
- index.mapper.dynamic 설정이 false면 true로 변경합니다.

표 4-1 artists 타입의 필드 구조

필드	설명
number	문서 ID 값

⁰³ 색인 필드의 데이터에 대한 데이터 타입을 지정하지 않고 동적으로 Elasticsearch에서 적절한 타입을 지정하는 기능이다(<http://bit.ly/1B9MQ78>).

필드	설명
name	아티스트 이름
url	아티스트 웹 사이트
picture	아티스트 사진
@timestamp	date 타입으로 매핑되도록 "@"를 붙여서 선언

텍스트 형태의 파일뿐만 아니라 JSON 형태의 파일도 색인할 수 있습니다. 메인 메서드에서 '// text' 아래 있는 코드를 다음 코드로 바꿔 실행하면 정상적으로 동작합니다.

[JSON 파일 구조 – IndexWriter.java]

```
Job json = new Job(job.getConfiguration());
TextInputFormat.addInputPath(json, new Path("data/artists.json"));
json.setMapperClass(Mapper.class);
json.setMapOutputValueClass(Text.class);
json.getConfiguration().set(ConfigurationOptions.ES_INPUT_JSON, "true");

boolean result = json.waitForCompletion(true);
```

JSON 구조는 텍스트와 다르게 TabMapper와 같은 별도 mapper 클래스가 필요 없고 기본 mapper 클래스를 등록해서 기능을 구현할 수 있습니다. 이는 Elasticsearch에서 색인하기 위한 BulkRequest의 JSON 문서 구조가 이미 포함되어 있어 라인 단위로 읽고 처리하기 때문입니다.

TabMapper는 템 구분자가 있는 텍스트 문서를 색인할 때 필요한 mapper 클래스입니다. 이 클래스는 Elasticsearch 하둡 플러그인 통합 테스트 코드 (AbstractMRNewApiSaveTest.java)에서 작성한 코드와 동일합니다. 여기서는 맵리듀스로 색인 프로그램을 작성할 때 mapper 클래스 영역을 요구사항에 따라 작성하므로 전체 코드를 소개합니다.

[TabMapper 클래스]

```
public static class TabMapper extends Mapper {
```

```

@Override
public void map(Object key, Object value, Context context) throws IOException,
InterruptedException {
    StringTokenizer st = new StringTokenizer(value.toString(), "\t");
    Map<String, String> entry = new LinkedHashMap<String, String>();

    entry.put("number", st.nextToken());
    entry.put("name", st.nextToken());
    entry.put("url", st.nextToken());

    if (st.hasMoreTokens()) {
        String str = st.nextToken();
        if (str.startsWith("http")) {
            entry.put("picture", str);
            if (st.hasMoreTokens()) {
                String token = st.nextToken();
                entry.put("@timestamp", token);
            }
        }
        else {
            entry.put("@timestamp", str);
        }
    }
    context.write(key, WritableUtils.toWritable(entry));
}
}

```

4.1.3 검색 MapReduce 구현

검색용 맵리듀스를 작성하기 위해 IndexSearcher라는 메인 메서드가 있는 클래스를 작성합니다. IndexSearcher는 Elasticsearch로 별도의 분산 질의를 구성하지 않습니다. 이는 Elasticsearch 내부에서 샤프트로 분산 질의가 이미 수행되기 때문입니다. 따라서 별도의 reducer 구현 없이 mapper만 구현합니다.

다음 코드는 Elasticsearch 하둡 플러그인의 맵리듀스 통합 테스트 코드를 이용하여 독립적으로 실행되도록 작성하였습니다. 하둡을 설치할 필요 없이 이를립스에서 로컬 맵리듀스로 동작하고, 검색 결과를 별도로 처리하기 위

해 SearchMapper를 등록합니다. 검색 질의 MR의 search_type은 scan이며 scroll 방식의 size 만큼의 문서를 가져옵니다.

[IndexSearcher.java – 메인 메서드]

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    conf.setBoolean("mapred.map.tasks.speculative.execution", false);
    conf.setBoolean("mapred.reduce.tasks.speculative.execution", false);
    conf.set(ConfigurationOptions.ES_NODES, "localhost:9200");
    conf.set(ConfigurationOptions.ES_RESOURCE, "radio/artists");
    HadoopCfgUtils.setGenericOptions(conf);

    Job job = new Job(conf);
    job.setInputFormatClass(EsInputFormat.class);
    job.setOutputKeyClass(Text.class);

    job.setOutputFormatClass(PrintStreamOutputFormat.class);
    job.setOutputKeyClass(Text.class);
    boolean type = random.nextBoolean();
    Class<?> mapType = (type ? MapWritable.class : LinkedMapWritable.class);
    job.setOutputValueClass(mapType);

    conf.set(ConfigurationOptions.ES_QUERY, "{ \"query\" : { \"match_all\" : {} } }");
    job.setMapperClass(SearchMapper.class);
    job.setNumReduceTasks(0);

    boolean result = job.waitForCompletion(true);
}
```

다음 코드는 검색 결과를 mapper에서 받아 처리합니다. 첫 번째 파라미터 key는 문서의 _id 값이고, 두 번째 파라미터 value는 문서의 데이터가 됩니다. 문서에 대한 전체 key 필드는 doc.keySet, 필드별 값은 doc.values에서 확인할 수 있습니다.

Mapper에서 사용한 `LinkedMapWritable`은 `MapWritable`을 이용하여 Elasticsearch 타입으로 변환할 수 있게 해줍니다.

```

public static class SearchMapper extends Mapper {
    @Override
    public void map(Object key, Object value, Context context) throws IOException,
    InterruptedException {
        Text docId = (Text) key;
        LinkedMapWritable doc = (LinkedMapWritable) value;

        System.out.println(docId);
        System.out.println(value);
        System.out.println(doc.keySet());
        System.out.println(doc.values());
    }
}

```

다음은 IndexSearcher의 실행 결과입니다.

실행 결과

```

# docId
uYAx5JYMRcyit5YC33bKrQ

# value
{number=790, name=Belinda, url=http://www.last.fm/music/Belinda, picture=http://
userserve-ak.last.fm/serve/252/62111231.png, @timestamp=2745-10-06T19:20:25.000Z}

# doc.keySet()
[number, name, url, picture, @timestamp]

# doc.values()
[790, Belinda, http://www.last.fm/music/Belinda, http://userserve-ak.last.fm/
serve/252/62111231.png, 2745-10-06T19:20:25.000Z]

```

하둡과 Elasticsearch의 데이터 탑입 매핑은 다음 표를 참고하기 바랍니다.

표 4-2 Hadoop과 Elasticsearch의 데이터 탑입 매핑

Writable	Elasticsearch 탑입
null	null
NullWritable	null
BooleanWritable	boolean

Writable	Elasticsearch 타입
Text	string
ByteWritable	byte
IntWritable	int
VInt	int
LongWritable	long
VLongWritable	long
BytesWritable	binary
DoubleWritable	double
FloatWritable	float
MD5Writable	string
ArrayWritable	array
AbstractMapWritable	map
UTF8 (Hadoop 1.x)	string
ShortWritable (Hadoop 2.x)	short

4.2 Hive 연동

Hive는 하둡의 데이터 웨어하우스 시스템 Data Warehouse System 으로, 저장된 데이터를 쉽게 분석, 질의, 요약할 수 있게 해주고, 이를 위해 SQL(HiveQL)과 다양한 UDF를 제공합니다.

Hive 설치 방법은 아파치 Hive 공식 문서인 [GettingStarted](#)⁰⁴를 참고하길 바라며 여기서는 설치 관련 내용은 다루지 않습니다.

4.2.1 준비 항목

Hive를 이용하려면 맵리듀스 구현 시 필요한 준비 항목에 추가로 Hive의

04 <http://bit.ly/1smMDeN>

metastore_db 저장 프로그램, 하둡, Hive 데몬이 실행되어야 합니다.

Hive 실행 시 elasticsearch-hadoop-VERSION.jar 파일이 Hive class path에 설정되어 있어야 합니다. 여기서는 Hive 0.12.0과 metastore_db로 MySQL 5.6.11을 사용하였습니다.

CLI 설정

```
$ bin/hive --auxpath=/path/elasticsearch-hadoop-VERSION.jar  
# or  
$ bin/hive -hiveconf hive.aux.jars.path=/path/elasticsearch-hadoop-VERSION.jar
```

hive-site.xml 설정

```
<property>  
    <name>hive.aux.jars.path</name>  
    <value>/path/elasticsearch-hadoop-VERSION.jar</value>  
    <description>A comma separated list (with no spaces) of the jar files</description>  
</property>
```

4.2.2 색인 구현

Hive를 이용한 색인 기능 구현은 매우 쉽습니다. 기본적으로 Hive에서 제공하는 JDBC 드라이버를 가지고 SQL문만 작성하면 구현됩니다. Indexer라는 메인 메서드가 있는 클래스를 작성하여 색인 기능을 구현해 보겠습니다. Hive의 external 테이블을 이용하여 데이터를 색인합니다.

맵리듀스에서 사용한 스키마와 데이터를 그대로 Hive에서도 사용하였습니다. 맵리듀스에서는 동적 매핑으로 색인이 이루어지지만, Hive로 색인 기능을 구현할 때는 CREATE문으로 스키마를 정의하고 테이블을 생성할 수 있습니다.

다음 코드는 load data 명령어를 사용하여 기초 데이터를 Hive 관리 테이블로 등록하고, Elasticsearch로 색인을 하기 위해 external 테이블을 생성합니다.

즉, ‘artists_hive’라는 테이블에 ‘LOAD DATA LOCAL INPATH …’를 통해 실제 HDFS에 데이터를 등록합니다.

[Indexer.java – 메인 메서드]

```
public static void main(String[] args) throws SQLException {
    ...중략...
    res = stmt.executeQuery("LOAD DATA LOCAL INPATH 'data/artists.dat' INTO TABLE
    artists_hive");
    res = stmt.executeQuery("CREATE EXTERNAL TABLE artists ("+
        "    id      BIGINT,"+
        "    seq     BIGINT,"+
        "    name   STRING,"+
        "    links  STRUCT<url:STRING, picture:STRING>,"+
        "    ts     STRING) "+
        "STORED BY 'org.elasticsearch.hadoop.hive.EsStorageHandler' "+
        "TBLPROPERTIES('es.nodes' = 'localhost:9200', " +
        "    'es.port' = '9200', " +
        "    'es.resource' = 'radio/artists', " +
        "    'es._id.field' = 'id', " +
        "    'es.mapping.timestamp' = 'ts', " +
        "    'es.index.auto.create' = 'true')");
    ...중략...
```

앞의 코드는 인덱스와 탑업 생성까지만 다루었고, 이제 실제 데이터를 등록하는 과정을 살펴보겠습니다. 다음 코드는 external 테이블을 통해 Elasticsearch로 인덱스와 탑업을 구성하였고, INSERT문으로 실제 HDFS에 등록된 데이터를 Elasticsearch로 색인하는 역할을 수행합니다.

[데이터 등록]

```
sql = "INSERT OVERWRITE TABLE artists SELECT s.seq, s.seq, s.name, named_struct( 'url' ,
    s.url, 'picture', s.picture), s.ts FROM artists_hive s";
res = stmt.executeQuery(sql);
...중략...
```

Hive를 이용한 색인 과정은 다음과 같습니다.

- **HDFS로 데이터 등록** : 색인에 필요한 원본 데이터를 하둡 셸(Hadoop shell) 또는 소프트웨어를 이용하여 등록합니다.
- **Hive external 테이블 생성** : Elasticsearch의 인덱스와 탑입을 생성합니다.
- **Hive의 insert overwrite로 external 테이블에 데이터 등록** : Elasticsearch로 색인합니다.

Hive와 Elasticsearch의 데이터 탑입 매핑은 다음 표를 참고하기 바랍니다.

표 4-3 Hive와 Elasticsearch의 데이터 탑입 매핑

Hive 탑입	Elasticsearch 탑입
void	null
boolean	boolean
tinyint	byte
smallint	short
int	int
bigint	long
double	double
float	float
string	string
binary	binary
timestamp	date
struct	map
map	map
array	array
union	not supported
decimal (0.11 이상)	string
date (0.12 이상)	date
varchar (0.12 이상)	string
char (0.13 이상)	string

Elasticsearch 하둡 플러그인을 사용할 때도 설정 관련 파라미터를 최적화할 수 있습니다. 자세한 내용은 [표 4-4]를 참고하길 바랍니다.

TBLPROPERTIES 알아보기

하둡에서 주로 사용하는 configure의 Elasticsearch용 property를 Elasticsearch에서는 TBLPROPERTIES라고 합니다. 이 정보는 Elasticsearch Map Reducer 코드 작성 시 설정 정보 지정에 필요한 property이므로 옵션을 간단히 살펴보겠습니다.

표 4-4 Configuration 옵션⁰⁵

Property	설명
es.host	Elasticsearch 노드의 호스트 정보 지정
es.nodes	Elasticsearch 클러스터 노드의 정보 지정
es.port	es.host로 연결할 HTTP 포트 지정
es.batch.*	Bulk 요청 설정 정보 지정
es.http.*	HTTP 연결/재시도 설정 정보 지정
es.scroll.*	Elasticsearch 검색 타입 scroll 설정 정보 지정
es.ser.*	Serialization settings으로 JSON과 bytes 처리를 위한 설정
es.input.json	입력 데이터 타입 설정 (no → 텍스트, yes → JSON)
es.index.*	인덱스 생성 또는 missing에 대한 settings 설정
es.mapping.*	매핑 타입 설정
es.write.operation	Elasticsearch 연산(index, create, update, upsert, delete) 타입 설정
es.update.*	Elasticsearch 업데이트 스크립트 연산 설정
es.net.*	Elasticsearch 네트워크 옵션 설정

4.2.3 검색 구현

Hive JDBC 드라이버를 이용하여 검색하기 위한 Searcher 애플리케이션을 구현해 보겠습니다. Searcher는 HiveQL을 이용하여 Elasticsearch로 검색 질의를 하는데, 내부적으로는 Hive의 external 테이블을 이용하여 Elastic

05 ConfigurationOptions.java 참고: <http://bit.ly/1NmVSTs>

search에 저장된 데이터에 대한 질의 결과를 매핑하여 결과를 반환합니다. 즉, 실제 데이터는 Elasticsearch에 있고, external 테이블에서는 메타 데이터만 사용합니다.

Hive CLI를 이용해서 external 테이블이 생성된 것을 확인할 수 있고, external 테이블로 질의를 실행해서 결과를 확인할 수도 있습니다.

Hive CLI를 이용한 테이블 확인

```
hive> show tables;  
OK  
artists  
artists_hive  
Time taken: 0.09 seconds, Fetched: 3 row(s)
```

[Searcher.java - 메인 메소드]

```
public static void main(String[] args) throws SQLException {  
    ...중략...  
    Statement stmt = con.createStatement();  
    ResultSet res;  
    res = stmt.executeQuery("DROP TABLE artists");  
    res = stmt.executeQuery("CREATE EXTERNAL TABLE artists ("+  
        "    id      BIGINT,"+  
        "    seq      BIGINT,"+  
        "    name     STRING,"+  
        "    links    STRUCT<url:STRING, picture:STRING>) "+  
        "STORED BY 'org.elasticsearch.hadoop.hive.EsStorageHandler' "+  
        "TBLPROPERTIES('es.nodes' = 'localhost:9200', 'es.port' = '9200', 'es.resource' =  
    'radio/artists', " +  
        " 'es.query' = '?q=name:b*')");  
  
    sql = "select t2.* from artists t1 join artists_hive t2 on ( t2.seq = t1.id )";  
    res = stmt.executeQuery(sql);  
  
    while (res.next()) {  
        System.out.println(String.valueOf(res.getInt(1)) + "\t"  
            + res.getString(2) + "\t"  
            + res.getString(3) + "\t"  
            + res.getObject(4)
```

```
    );  
}  
...중략...
```

이 예제는 artists라는 external 테이블로 `q=name:b*`라는 질의를 실행하여 결과를 저장합니다. 실제 HDFS의 artists 경로에는 물리적인 데이터가 저장되어 있지 않으며, 실행 시점에 Elasticsearch로 질의하여 데이터를 가져옵니다. Elasticsearch 하둡 플러그인과 Hive의 external 테이블을 이용하면 이 예제에서와 같이 조인 기능을 쉽게 구현할 수 있습니다. 단, Elasticsearch에서는 Union 기능을 아직 제공하지 않으므로 유의해서 사용해야 합니다.

4.3 정리

이 장에서는 빅데이터 관점에서 검색엔진을 기능적으로 어떻게 활용할 수 있는지 살펴보았습니다. HDFS에 저장된 데이터를 빠르게 검색엔진으로 색인하고 검색 엔진에서 제공하는 분석 API(Aggregation)를 활용하면 목적에 맞는 분석 결과를 쉽게 얻을 수 있을 것으로 기대합니다. SQL on Hadoop 제품 도입이 쉽지 않다면 Elasticsearch 하둡 플러그인을 활용해 보길 추천합니다.

ELK 연동

Elasticsearch에서 제공하는 ELK 스택은 Elasticsearch, Logstash, Kibana로 구성되어 있습니다. Elasticsearch는 실시간 검색과 분석을 위한 도구고, Logstash는 데이터를 가공하기 위한 도구입니다. Kibana는 Logstash로 가공된 데이터를 Elasticsearch로 색인하여 그 결과로 각종 그래프와 도표를 통한 데이터 시각화 기능을 제공하는 도구입니다. ELK stack은 모두 오픈소스 기반이며 하나하나가 데이터를 처리하는 데 강력한 도구로 활용됩니다.

여기서는 ELK 스택을 연동하는 기본적인 내용을 다루겠습니다.

5.1 Logstash

Logstash는 모든 종류의 로그를 받고 처리하고 출력하는 도구로, 백엔드^{Back-end}의 데이터 저장소로는 Elasticsearch, 프론트엔드^{Front-end}의 데이터 시각화 도구로는 Kibana를 사용합니다. 이를 사용하려면 다음 JDK 버전이 필요합니다.

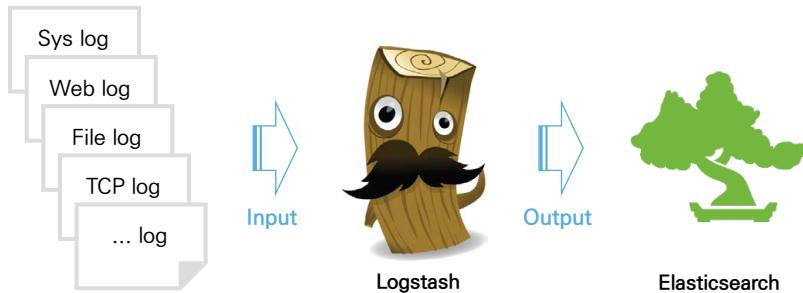
- JDK 1.7.X 이상 권장
- OpenJDK 또는 Oracle 버전 권장

JDK 버전 확인

```
$ java -version
java version "1.7.0_55"
Java(TM) SE Runtime Environment (build 1.7.0_55-b13)
Java HotSpot(TM) 64-Bit Server VM (build 24.55-b03, mixed mode)
```

Logstash는 동작 방식이 단순해서 사용하기 쉽고 프로그램과 연동하기 쉬우며, 다양한 로그 형태가 있습니다. Logstash의 기본 동작 방식은 다음과 같습니다.

그림 5-1 Logstash 작업 흐름



5.1.1 다운로드와 설치

다음 명령으로 Logstash를 내려받습니다. Logstash는 Elasticsearch를 기본으로 내장⁰¹하고 있습니다.

다운로드

```
$ curl -o https://download.elasticsearch.org/logstash/logstash/logstash-1.4.2.tar.gz
```

설치는 컴파일이나 설정 과정이 필요 없이 압축 해제만 하면 바로 사용할 수 있습니다.

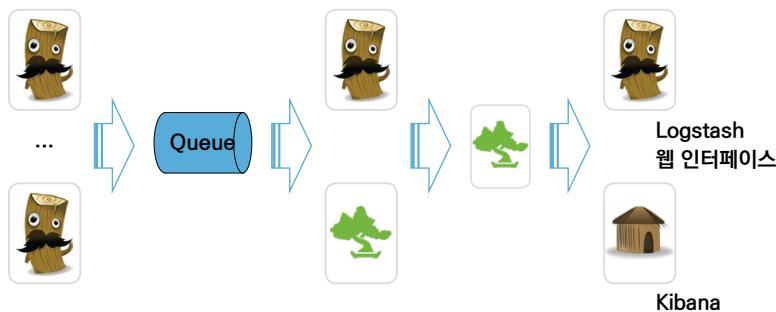
설치

```
$ tar -xvzf logstash-1.4.2.tar.gz
```

확장성을 고려한다면 다음 그림을 참고하여 아키텍처를 구성하기 바랍니다.

01 가장 최신 버전은 현재 1.4.2이고 내장된 Elasticsearch 버전은 1.1.1이다. Logstash 1.5부터는 Elasticsearch 1.4.0이 포함될 예정이다.

그림 5-2 Logstash 아키텍처



Log Collector Log 임시 저장소 Log Indexer Log 저장 & 검색

- Collector 역할은 Logstash 외 다른 도구를 사용해도 괜찮습니다.
- Queue는 전형적인 메시지 큐 소프트웨어를 사용해도 되고, Redis, MongoDB 같은 NoSQL 종류를 사용해도 됩니다.
- Indexer는 Logstash의 입/출력 정의로 색인할 수 있으며, Elasticsearch의 River 플러그인으로도 구성할 수 있습니다.
- 색인이 완료되면 Logstash의 웹 인터페이스나 Kibana로 쉽게 분석과 조회를 할 수 있습니다.

5.1.2 실행과 테스트

실행은 아주 쉽게 진행할 수 있습니다. 다음은 콘솔에서 표준 입력(stdin)으로 받아 표준 출력(stdout)으로 출력하는 예제입니다. 실행 후 중지는 CTRL-C 명령어로 빠져나오면 됩니다.

실행 - 표준입/출력

```
$ bin/logstash -e 'input { stdin { } } output { stdout { } }'  
logstash input output test  
2014-12-04T06:15:15,112+0000 jeong-ui-MBP logstash stdin/stdout test
```

다음은 Codec을 이용하여 출력 형식을 정의한 예제입니다. 입력 문자열에 대한 출력은 JSON 구조로 나온 것을 확인할 수 있습니다. 코드에서 굵게 표시된 부분이 입력 문자열입니다.

실행 - Codec을 이용한 출력

```
$ bin/logstash -e 'input { stdin { } } output { stdout { codec => json } }'  
logstash ouput codec is json.  
{ "message": "logstash ouput codec is json.", "@version": "1", "@timestamp": "2014-12-  
04T06:24:41.347Z", "host": "jeong-ui-MBP" }
```

다음은 입력 로그를 Elasticsearch로 색인하는 예제입니다. 굵게 표시된 부분이 입력 문자열이며, 출력은 Elasticsearch로 색인되어 있는 것을 확인할 수 있습니다. 출력값으로 지정한 Elasticsearch의 config 설정값들은 [5.1.4 Config 알아보기](#)에서 자세히 다루겠습니다.

실행 - Elasticsearch를 이용한 출력

```
$ bin/logstash -e 'input { stdin { } } output { elasticsearch { action => index host =>  
localhost port => 9300 } }'  
input test1  
input line test2
```

Elasticsearch로 색인되는 문서의 매핑 구조는 다음과 같으며, 동적 템플릿 Dynamic Template⁰²을 통해서 생성됩니다.

- **Message** : 입력 로그의 문자열
- **@version** : 현재 스키마 버전으로, 기본값은 1
- **@timestamp** : 로그 이벤트가 발생한 시간
- **Host** : 입력 로그가 발생한 호스트 정보

5.1.3 Command-line Flag 알아보기

Logstash는 에이전트 Agent와 웹 형태로 사용할 수 있는데, 실행에 필요한 커맨드

02 탑업 매핑을 사전에 기술하지 않고 동적 매핑에 의해 정의되는 시점에 동적으로 탑업 매핑 구성하는 기능이다. 즉, 매핑 탑업을 미리 선언하지 않고 패턴이나 분석 특성에 맞춰 구성하는 것을 말한다(<http://bit.ly/18WaWYv>).

라인 플래그를 유형별로 살펴보겠습니다. 커マン드라인 플래그 정보는 `--help` 또는 `-h` 옵션으로 확인할 수 있습니다.

에이전트에서 사용하는 플래그는 다음과 같습니다.

표 5-1 에이전트의 플래그 종류

플래그	설명
<code>-f, --config CONFIGFILE</code>	에이전트 설정 정보를 파일로 받습니다.
<code>-e CONFIGSTRING</code>	에이전트 설정 정보를 문자열(string)로 받습니다.
<code>-w, --filterworkers COUNT</code>	에어전트에서 필터 기능을 수행할 worker 크기를 정의합니다.
<code>--watchdog-timeout TIMEOUT</code>	필터가 응답하지 않을 때 Logstash를 중지하기 위한 시간을 설정합니다.
<code>-l, --log FILE</code>	로그를 출력할 파일을 지정합니다(없을 경우 표준 출력).
<code>--verbose</code>	Info 수준의 자세한 로그를 출력합니다.
<code>--debug</code>	디버그 수준의 자세한 로그를 출력합니다.
<code>--pluginpath PLUGIN_PATH</code>	플러그인을 찾을 수 있는 경로를 설정합니다.

실행

```
$ bin/logstash --help
```

웹 인터페이스는 내부적으로 Kibana를 사용합니다.

표 5-2 웹의 플래그 종류

플래그	설명
<code>-a, --address ADDRESS</code>	웹 서버의 IP 주소를 설정합니다(기본 0.0.0.0).
<code>-p, --port PORT</code>	웹 서버의 포트를 설정합니다(기본 9292).

실행

```
$ bin/logstash-web --help
```

5.1.4 Config 알아보기

Logstash가 동작하는 데 필요한 설정으로, Elasticsearch에서는 Logstash Config Language라고 부릅니다. 이는 Logstash를 쉽고 간결하게 이용할 수 있게 도와줍니다.

Config Language는 크게 세 부분으로 나뉘며 구성은 다음과 같습니다.

```
input {  
    ...  
}  
  
filter {  
    ...  
}  
  
output {  
    ...  
}
```

주석

Config 작성 시 주석은 다음과 같이 작성합니다.

```
# 주석  
STRUCTURE { # 주석  
    # 주석  
}
```

플러그인

로그에 대한 필터, 입/출력 처리 설정으로 형식은 다음과 같습니다.

- 로그 입력은 stdin으로 받습니다.
- stdin으로 받은 로그가 필터에서는 source로 정의되며 message 필드에 저장되는 문자열을 가지고 필터 작업을 합니다.
- add_field는 말 그대로 필드를 추가하라는 의미입니다.

- 추가 필드명은 filter_{varField}이고 {varField}에서 변수명은 varField입니다. 필드의 값이 'hello'라면 최종 추가 필드명은 filter_hello가 됩니다.
 - 출력은 Elasticsearch로 색인하도록 구성되어 있습니다.
-

```
input {
    stdin { }
}

filter {
    json {
        source => "message"
        add_field => {
            "filter_{varField}" => "{host}) This is a filter field."
        }
    }
}

output {
    elasticsearch {
        action => index
        host => localhost
        port => 9300
    }
}
```

값의 타입

Config Language에서 사용하는 값의 타입은 다음과 같습니다.

- **Boolean** : 반드시 큰따옴표(" ")가 없는 true 또는 false 값을 가집니다.
-

```
FIELD => true
FIELD => false
```

- **String** : 단일값의 문자열이어야 합니다.
-

```
FIELD => "문자열"
```

- **Number** : 정수나 부동소수점을 갖는 유효한 숫자여야 합니다.
-

```
FIELD => 123  
FIELD => 123.01
```

- **Array** : 단일값의 문자열이나 여러 개의 문자열 배열이어야 합니다.
-

```
FIELD => "문자열"  
FIELD => ["문자열", "문자열"]
```

- **Hash** : Logstash는 Ruby라는 언어로 개발되어서 기본적으로 Ruby의 Hash와 같다고 보면 됩니다.
-

```
FIELD => {  
    "key_field1" => "value1"  
    "key_field2" => "value2"  
    ...  
}
```

조건 처리

Logstash를 사용하다 보면 필터나 출력 영역에서 어떤 조건에 따른 처리가 필요할 때가 있는데, 이 처리를 위해서 if, else if, else 구문을 제공합니다.

```
if CONDITION {  
    ...  
} else if CONDITION {  
    ...  
} else {  
    ...  
}
```

5.1.5 Input 알아보기

Logstash로 로그 데이터를 전달하기 위한 방법을 기술하는 것으로, 다음 플러그인이 일반적으로 유용하게 많이 사용됩니다.

File

파일 시스템의 파일로부터 로그 데이터를 읽어 들이는 방식으로, tail -0a와 비슷합니다. path는 필수 입력값이고 나머지는 옵션 값입니다.

```
input {  
    file {  
        path => "/home/elasticsearch/logs/*.log"  
        type => "eslog"  
    }  
}
```

Syslog

Syslog에서 TCP와 UDP에 대한 리스너가 실행되고 Syslog 메시지를 읽어와 RFC3164에 의해 파싱합니다. Property 지정이 없으면 기본값으로 모든 옵션 값이 구성됩니다.

```
input {  
    syslog {  
        host => "0.0.0.0"  
        port => 514  
        type => "syslog"  
    }  
}
```

Redis

Redis 서버에서 로그 데이터를 읽어 들이는 방식으로, Redis는 종종 Logstash 를 위한 큐로도 활용됩니다. Property 지정이 없으면 기본값으로 모든 옵션 값이

구성이 됩니다. host, port, db, password는 Redis 서버의 연결 정보를 지정합니다.

```
input {
    redis {
        host => "localhost"
        port => 6379
        db => 0
        password => "xxxxxx"
        type => "redislog"
    }
}
```

Logstash는 이외에도 다양한 입력 플러그인⁰³을 제공하므로 자세한 내용은 홈페이지를 참고하기 바랍니다.

5.1.6 Filter 알아보기

필터는 Logstash로 전달된 로그 데이터의 중간 처리 과정에 사용되는데, 로그 데이터를 최종 출력으로 전달하기 전에 값을 조합하거나 제거하는 등의 작업에 유용합니다. 다음은 일반적으로 사용되는 필터 플러그인입니다.

Grok

이 도구는 Logstash에서 사용하는 비구조화된 데이터를 처리하는 데 가장 좋은 대안입니다. 그리고 syslog, apache, mysql 등의 로그를 일반적인 로그 형식으로 완벽하게 처리할 수 있습니다.

```
filter {
    grok {
        match => ["message", "%{IP:client} %{WORD:method} %{URIPATHPARAM:request}"]
```

03 <http://bit.ly/1CEqf4H>

```
    }  
}
```

다음 형식은 HTTP request 로그를 기준으로 작성되었습니다.

```
"192.168.1.100 GET /index.html 1024 0.012"
```

- %{IP:client}는 client라는 필드에 IP형의 데이터(192.168.1.100)를 가진다는 의미입니다.
- %{WORD:method}는 method라는 필드에 WORD 형의 데이터(GET)를 가진다는 의미입니다.
- %{URIPATHPARAM:request}는 request라는 필드에 URIPATHPARAM 형의 데이터(/index.html)를 가진다는 의미입니다.

여기서 표현한 데이터형은 모두 특정 패턴을 가지며 logstash/patterns 폴더 아래 grok-patterns에 정의되어 있습니다.

Mutate

이 필터는 로그 데이터의 필드(Add, rename, remove, replace, update 등)를 변형할 수 있게 해줍니다. 입력 로그 데이터에서 country라는 필드에 해당하는 값을 이용하여 client_%{country}라는 필드가 추가되고, 여기에 클라이언트의 IP 값이 들어갑니다.

```
filter {  
    mutate {  
        add_field => { "client_%{country}" => "%{ip}" }  
    }  
}
```

Drop

어떤 조건과 조합해서 일치하면 로그 데이터를 drop(제거 또는 배제)하는 필터입니다.

다. 다음은 loglevel이 debug일 때 모든 데이터를 drop합니다.

```
filter {
    if [loglevel] == "debug" {
        drop { }
    }
}
```

Logstash는 이외에도 다양한 필터 플러그인⁰⁴을 제공하므로 자세한 내용은 홈페이지를 참고하기 바랍니다.

5.1.7 Output 알아보기

Logstash 처리 과정의 마지막 단계로 로그 데이터의 최종 배치가 완료되어 이벤트 실행이 완료됩니다. 이때 일반적으로 사용되는 출력 타입은 다음과 같습니다.

Elasticsearch

Logstash의 가장 대표적인 출력으로, 로그 데이터를 Elasticsearch로 저장합니다. 이를 이용하여 Query DSL로 편리하게 질의할 수 있습니다. 또한, Kibana를 이용하면 대시보드^{Dashboard} 관리와 저장된 데이터 분석이 쉽습니다.

Elasticsearch의 최소 버전은 1.0.0 이상이어야 하고 필수 항목은 없으며, 지정된 값이 없으면 기본값으로 동작합니다. 기본 Elasticsearch 정보를 등록하고 지정한 인덱스와 타입으로 색인합니다.

```
output {
    elasticsearch {
        cluster => "elasticsearch"
        host => localhost
        port => 9300
```

04 <http://bit.ly/1Fh8V37>

```
        action => "index"
        index => "logstash-%{+_YYYY.MM.dd}"
        index_type => "log"
    }
}
```

File

로그 데이터는 최종으로 디스크에 파일로 쓰는데, 이때 필수로 path 정보가 등록되어야 합니다. 로그 데이터는 GZIP으로 압축하고 해당 path에 기록합니다.

```
output {
    file {
        gzip => true
        path => "/disk/logs/logstash-%{+_YYYY.MM.dd}.log"
    }
}
```

Statsd

UDP 또는 TCP를 통해 카운터나 데이터 등의 통계 집계를 수신하고 Graphite나 Datadog 플러그인 같은 백엔드 서비스로 집계를 보내는 네트워크 데몬입니다. 즉, WAS^{Web Application Server}의 응답 시간을 Graphite 서버로 전송하는 역할을 합니다.

```
output {
    statsd {
        host => "graphite.localhost"
        increment => "tomcat.response.%{response}"
    }
}
```

Logstash에서는 이외에도 다양한 출력 플러그인⁰⁵을 제공하므로 자세한 내용은 홈페이지를 참고하기 바랍니다.

5.1.8 Codec 알아보기

코덱은 입/출력 스트림에 대한 필터 역할을 하고, 로그 데이터를 쉽게 처리할 수 있게 도와줍니다. 가장 많이 사용하는 코덱으로는 JSON, MessagePack, Plain(text)이 있습니다.

JSON

입력 JSON 로그 데이터에는 복호화(decode) 작업을 수행하고, 로그 데이터의 출력 결과에는 암호화(encode) 작업을 수행합니다. 즉, JSON 형태의 로그 데이터 처리에 사용하는 코덱이라고 보면 됩니다. 입/출력에 모두 사용할 수 있고, 기본 캐릭터셋은 UTF-8입니다.

```
input {
    file {
        codec => json {
            charset => "UTF-8"
        }
    }
}
```

Msgpack(MessagePack)

JSON과 비슷하게 직렬화^{Serialization}와 역직렬화^{Deserialization} 과정을 수행합니다. 전달로그 데이터의 형식을 지정할 수 있고, sprintf 문자열을 지원합니다.

```
input {
    file {
```

⁰⁵ <http://bit.ly/18n9WeR>

```
    codec => msgpack {  
      format => ... # string (optional), default: nil  
    }  
  }  
}
```

Plain

이미 정의된 형태의 로그 데이터를 처리하는 데 유용한 코덱입니다. 캐릭터셋과 형식을 지정할 수 있습니다.

```
input {  
  file {  
    codec => plain {  
      charset => "UTF-8"  
      format => ... # string (optional)  
    }  
  }  
}
```

5.2 Elasticsearch

이 부분은 [기본편](#)⁰⁶에서 전반적인 내용을 살펴보았으므로 여기서는 버전 업그레이드에 따른 기본 설치와 확인해야 할 사항만 가볍게 다루겠습니다.

5.2.1 다운로드와 설치

[홈페이지](#)⁰⁷에서 Elasticsearch를 내려받아 설치합니다. 다운로드가 완료되면 압축을 해제합니다.

06 <http://www.hanbit.co.kr/ebook/look.html?isbn=9788968486913>

07 <http://www.elasticsearch.org/downloads/>

다운로드와 설치

```
$ wget https://download.elasticsearch.org/elasticsearch/elasticsearch/elasticsearch-1.3.6.tar.gz  
$ tar -xvzf elasticsearch-1.3.6.tar.gz
```

ELK 연동을 위해 버전을 맞춰 줍니다.

- Logstash 1.4.2
- Elasticsearch 1.3.X
- Kibana 3.1.X

5.2.2 실행과 테스트

Foreground 또는 Background를 선택하여 실행합니다.

실행

```
$ bin/elasticsearch  
# 또는  
$ bin/elasticsearch -d
```

테스트

```
$ curl -XGET http://localhost:9200/
```

5.2.3 기본 플러그인 설치

- **Head** : 이 플러그인은 Elasticsearch에 대한 기본 관리 UI를 제공합니다.

플러그인 설치

```
$ bin/plugin --install mobz/elasticsearch-head
```

- **Bigdesk** : 이 플러그인은 실시간으로 시스템 리소스 사용 현황을 조회할 수 있게 해줍니다.

플러그인 설치

```
$ bin/plugin --install lukas-vlcek/bigdesk
```

- **Sense** : 크롬 브라우저를 이용하여 Elasticsearch로 Query DSL을 작성하여 질의할 수 있게 해줍니다. 크롬 앱에서 링크⁰⁸를 통해 크롬 브라우저에 설치합니다.

5.3 Kibana

오픈소스 기반의 Kibana는 웹 브라우저에서 Elasticsearch를 위한 검색과 분석 대시보드Dashboard를 제공해 줍니다. 또한, HTML과 JavaScript로 구현되어 있어 별도로 응용 프로그램을 구현할 필요가 없습니다. 단, Kibana와 Elasticsearch 간 연동을 위해서는 Elasticsearch 설정에서 http.enabled가 true로 설정되어야 합니다. 여기서 사용할 데이터는 2장 검색 데이터 분석에서 사용한 예제 데이터를 기반으로 작성하였으니 참고하기 바랍니다.

5.3.1 다운로드와 설치

Kibana는 HTML과 JavaScript로 구현되어 있으므로 AWSApache Web Server나 톰캣Tomcat 같은 WASWeb Application Server에 배포해서 사용하길 추천합니다. 여기서는 톰캣을 이용해서 진행하겠습니다.

- **Kibana⁰⁹**

설치

```
$ wget https://download.elasticsearch.org/kibana/kibana/kibana-3.1.2.tar.gz  
$ tar -xvzf kibana-3.1.2.tar.gz
```

08 <http://bit.ly/1A2lhHN>

09 <http://bit.ly/17XdNzp>

- **Tomcat¹⁰** : 톰캣은 6.0.x, 7.0.x, 8.0.x 중 최신 릴리스 버전을 내려받으면 됩니다. 모든 버전에서 동일하게 동작하지만 여기서는 8.0.X를 내려받아 사용하겠습니다.

설치

```
$ wget http://apache.tt.co.kr/tomcat/tomcat-8/v8.0.15/bin/apache-tomcat-8.0.15.tar.gz  
$ tar -xvzf apache-tomcat-8.0.15.tar.gz
```

5.3.2 실행

압축을 해제한 Kibana 소스 파일을 톰캣의 webapps/ROOT 폴더로 복사 또는 이동한 후 톰캣을 실행합니다. ROOT 폴더에 있는 모든 파일은 Kibana를 사용할 때는 필요 없으므로 삭제합니다.

Kibana 복사

```
$ cd apache-tomcat-8.0.15/webapps/ROOT  
$ rm -rf *  
$ cp -rf ../../ kibana-3.1.2/* .
```

실행하기 전 Kibana에서 접속할 Elasticsearch 정보를 수정해야 합니다. 여기서는 기본 설정인 localhost:9200으로 접속하므로 별도로 수정하지 않지만, 필요하면 Elasticsearch 정보를 수정한 후 연동해야 합니다.

설정 변경

```
$ vi kibana-3.1.2/config.js  
+++++  
/** @scratch /configuration/config.js/1  
*  
* == Configuration  
* config.js is where you will find the core Kibana configuration. This file contains  
parameter that  
* must be set before kibana is run for the first time.
```

¹⁰ <http://tomcat.apache.org/>

```

*/
define(['settings'],
function (Settings) {
    ...중략...
    return new Settings({
        ...중략...
        elasticsearch: "http://" + window.location.hostname + ":9200",
        ...중략...
    });
});

```

Tomcat 실행

```

$ cd apache-tomcat-8.0.15
$ bin/startup.sh

```

실행을 중지하려면 bin/shutdown.sh를 실행합니다.

테스트하려면 웹 브라우저에서 <http://localhost:8080>으로 접속합니다. 다음 그림에서 보듯 초기 접속 화면은 기본적인 Kibana 소개만 제공됩니다. 화면 오른쪽 하단의 [1. Sample Dashboard]를 선택하면 현재 생성된 모든 인덱스의 기본 대시보드를 확인할 수 있습니다.

그림 5-3 Kibana 접속 화면

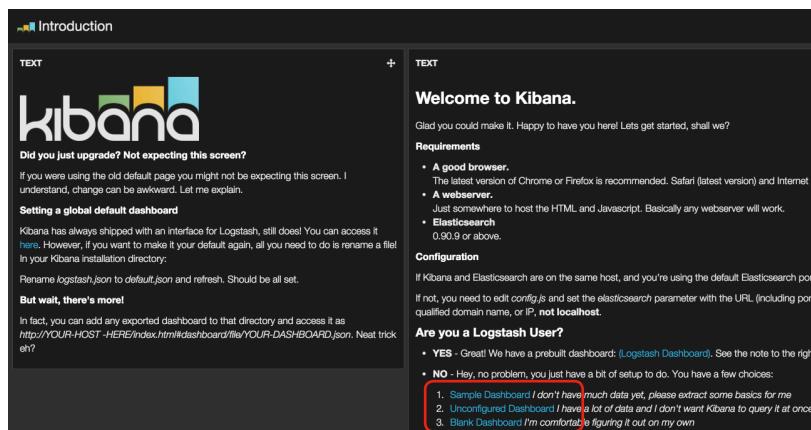


그림 5-4 샘플 대시보드 화면

5.3.3 대시보드 만들기

Kibana 초기 화면에서 [3. Blank Dashboard]를 선택합니다.

그림 5-5 대시보드 초기 화면

The screenshot shows the Grafana interface with a top navigation bar containing the URL 'localhost:8080/index.html#/dashboard/blank.json'. Below the navigation is a toolbar with various icons. The main area is titled 'New Dashboard' and contains a single chart placeholder labeled 'No filters available'. In the bottom right corner of the dashboard area, there is a red box highlighting a button labeled 'ADD A ROW'.

- 왼쪽 상단 블록 지정 부분의 [New Dashboard]는 대시보드 제목입니다.
 - 오른쪽 상단 블록 지정 부분은 대시보드 설정 버튼입니다.
 - 오른쪽 하단 블록 지정 부분의 [ADD A ROW]는 패널 panel 생성을 위한 행 추가 버튼입니다.

다시보드 설정

[그림 5-5]의 설정 버튼을 눌러 대시보드의 기본 설정을 합니다.

General

[Title]에 대시보드의 제목을 입력하고 [Save] 버튼을 눌러 저장합니다.

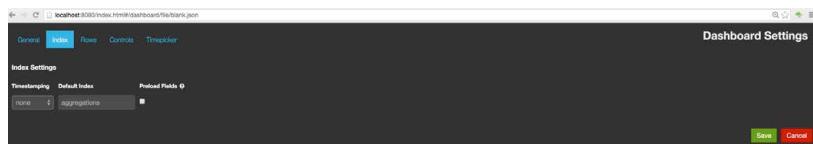
그림 5-6 대시보드 설정 - General



Index

이 영역에서는 일반적으로 Logstash와 연동하기 위해 timestamping을 설정하게 되지만 여기서는 2장 검색 데이터 분석에서 사용했던 aggregations 인덱스를 사용하겠습니다. [Timestamping]은 ‘none’으로 선택하고 [Default index]에는 ‘aggregations’를 입력합니다.

그림 5-7 대시보드 설정 - Index



Row 생성

앞의 과정까지가 기본 설정이며 이제부터 분석을 위한 패널을 추가하고 대시보드를 구성하겠습니다.

2장 검색 데이터 분석에서 생성한 문서를 기반으로 terms/stats aggregation 등을 테스트하기 위해 다음 필드 정보를 한 번 더 확인해 보겠습니다.

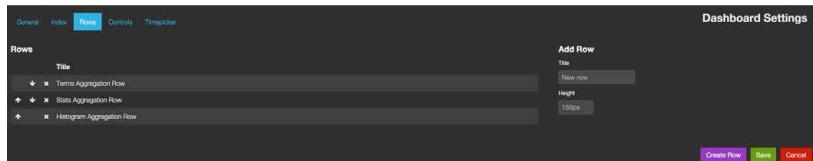
표 5-3 필드 정보

필드명	설명
item_code	상품코드
item_name	상품명
item_category	상품분류
buyer_id	구매자아이디
buyer_gender	구매자성별

필드명	설명
buyer_location	구매위치
buyer_country	구매국가
payment_price	결제금액
payment_type	결제유형
payment_time	결제시간

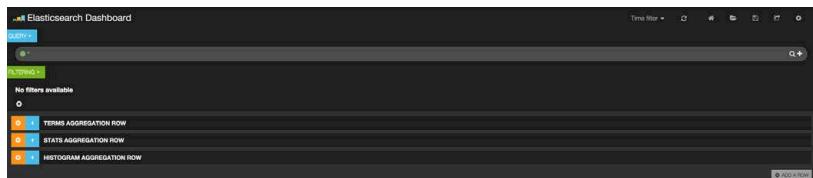
[Blank Dashboard] 화면의 [ADD A ROW] 버튼 눌러서 다음 화면이 나오면 대시보드에 구성할 aggregation 그룹을 생성하고 저장합니다.

그림 5-8 Aggregation Row 추가



그 다음 그룹별로 패널을 생성합니다.

그림 5-9 빈 Row



Terms Aggregation Panel

- **구매자 아이디별 통계**: row를 생성하고 해당 row에 패널을 생성합니다.

Step 1) [Add panel to empty row] 버튼 눌러 패널 생성

Step 2) [Select Panel Type]에서 'terms' 선택

Step 3) [Title]에 '구매자 아이디별 통계' 입력

Step 4) [Parameters]의 [Field] 항목에 'buyer_id' 입력

Step 5) [Save] 버튼을 눌러 설정 저장

그림 5-10 구매자 아이디별 통계 패널 생성

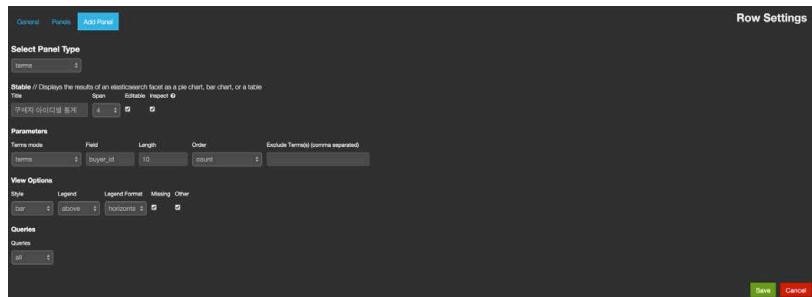
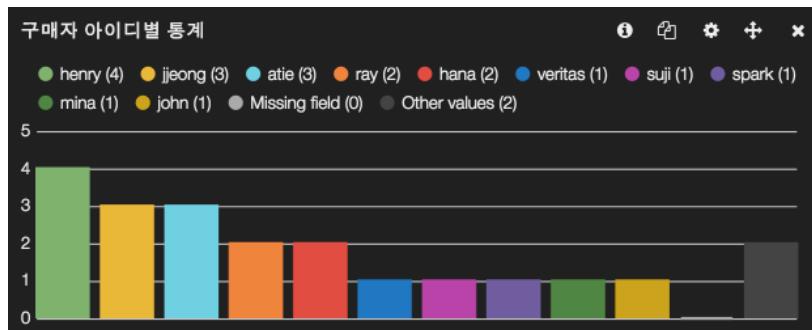
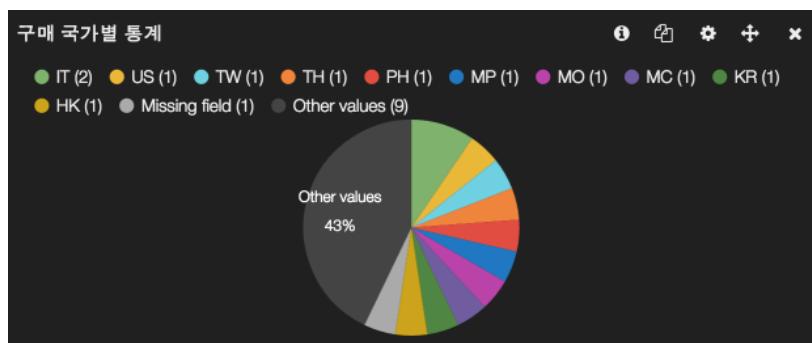


그림 5-11 구매자 아이디별 통계 패널 화면



- **구매 국가별 통계 :** 구매자 아이디별 통계 과정을 반복합니다. [Title]에는 ‘구매 국가별 통계’를 입력하고, [Parameters]의 [Field] 항목에는 ‘buyer_country’를 입력합니다. 여기서는 Pie 도표를 사용하므로 [View Options]의 [Style] 항목은 ‘pie’로 선택합니다.

그림 5-12 구매 국가별 통계 패널 화면



- 결제 유형별 통계 : 구매자 아이디별 통계 과정을 반복합니다. [Title]에는 ‘결제 유형별 통계’를 입력하고, [Parameters]의 [Field] 항목은 ‘payment_type’를 입력합니다. 여기서는 테이블을 사용하므로 [View Options]의 [Style] 항목은 ‘table’로 선택합니다.

그림 5-13 결제 유형별 통계 패널 화면

Term	Count	Action
card	6	🔍 ⚡
pay8	5	🔍 ⚡
bank	4	🔍 ⚡
phone	3	🔍 ⚡
cash	3	🔍 ⚡
Missing field	0	🔍 ⚡
Other values	0	🔍 ⚡

Stats Aggregation Panel

- 결제 금액 통계 : 생성된 row에 패널을 생성합니다.

Step 1) [Add panel to empty row] 버튼을 눌러 패널 생성

Step 2) [Select Panel Type]에서 ‘terms’ 선택

Step 3) [Title]에 ‘결제 금액 통계’ 입력

Step 4) [Details]에서 [Featured Stat] 항목은 ‘total’ 선택, [Field] 항목은 ‘payment_price’ 입력

Step 5) [Save] 버튼을 눌러 설정 저장

그림 5-14 결제 금액 통계 패널 생성

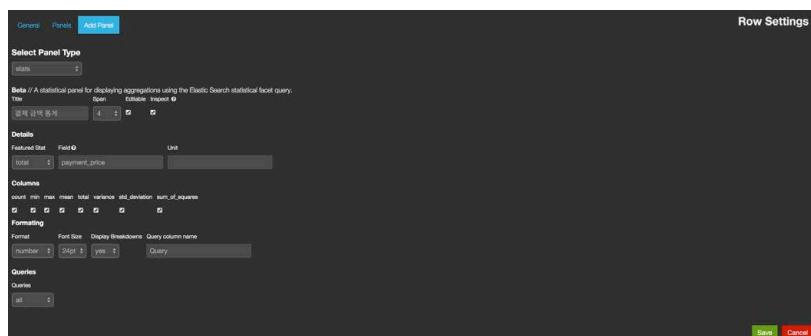


그림 5-15 결제 금액 통계 패널 화면



Histogram Aggregation Panel

- 결제일별 결제 금액 통계 : 생성된 row에 패널을 생성합니다.

Step 1) [Add panel to empty row] 버튼을 눌러 패널 생성

Step 2) [Select Panel Type]에서 'histogram' 선택

Step 3) [Title]에 '결제일별 결제 금액 통계' 입력

Step 4) [Values]에서 [Chart value] 항목은 'total' 선택, [Value Field] 항목은 'payment_price' 입력

Step 5) [Time Options]의 [Time Field] 항목에 'payment_time' 입력

Step 6) [Save] 버튼을 눌러 설정 저장

그림 5-16 결제일별 결제 금액 통계 패널 생성

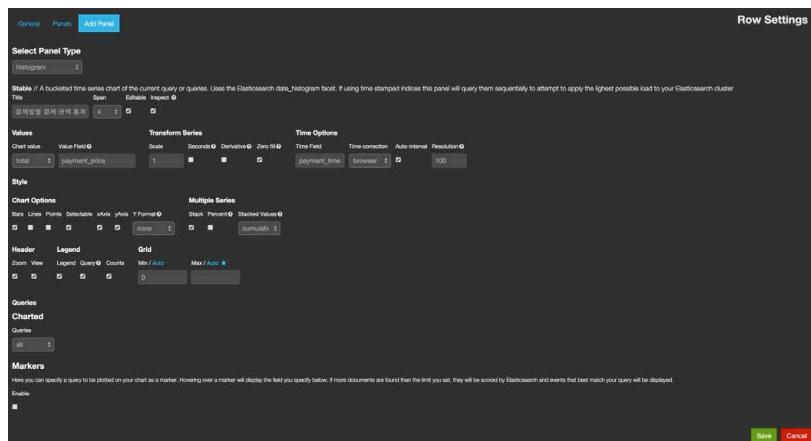
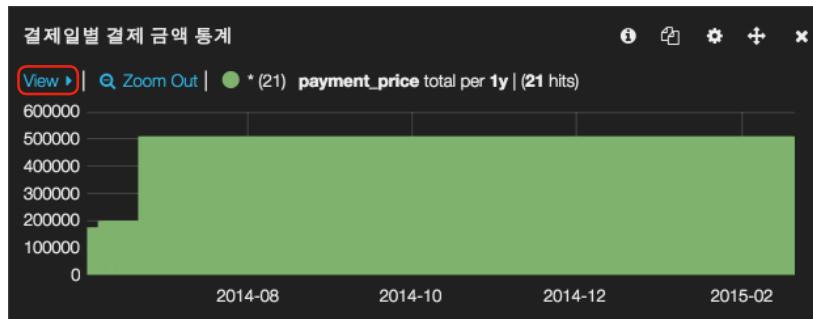


그림 5-17 결제일별 결제 금액 통계 패널 화면



앞의 결과는 1년(1y) 단위이므로 좀 더 세분화하여 결과를 만들어 보겠습니다. 블록된 부분의 [View] 버튼을 클릭하여 [Interval]을 1일(1d)로 조정합니다. 다음은 Interval이 1일일 때 결과 화면입니다.

그림 5-18 결제일별 결제 금액 통계 패널 화면 – Interval 1d



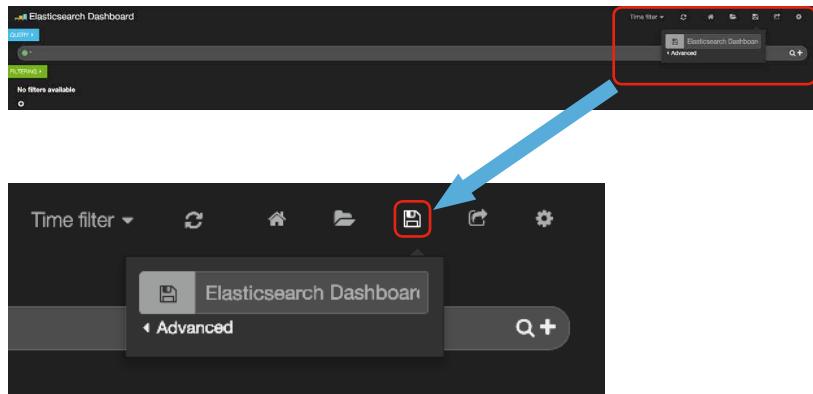
대시보드 저장과 로드

대시보드의 구성이 완료되면 저장합니다. 저장하지 않으면 설정한 정보가 유지되지 않으며 새로 생성해야 합니다. 저장과 로드 방법은 다음과 같습니다.

먼저 디스크 모양의 [저장] 버튼을 눌러 대시보드 제목을 입력하고 저장합니다. [Advanced] 영역에는 다음 3가지 항목이 있습니다.

- **Save as Home** : 현재 대시보드를 home으로 설정
- **Reset Home** : 설정된 home을 리셋
- **Export Schema** : 대시보드 스키마 정보를 JSON 파일로 다운로드

그림 5-19 대시보드 저장



그다음 폴더 모양의 [열기] 버튼을 누르면 다음 그림처럼 저장된 대시보드 목록이 보입니다. 이중에서 앞에서 생성한 'Elasticsearch Dashboard'를 선택합니다.

그림 5-20 대시보드 로드

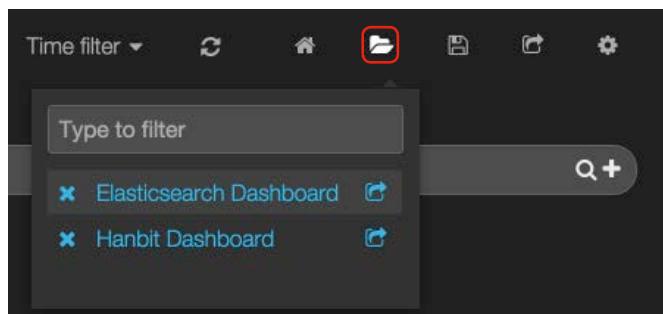
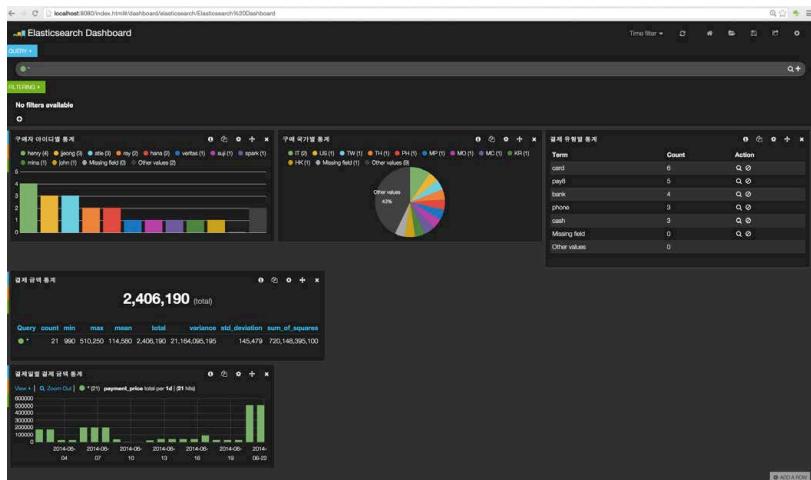


그림 5-21 Elasticsearch Dashboard



Kibana는 생성한 대시보드 정보를 Elasticsearch 안에 인덱스로 저장합니다. 이 인덱스는 kibana-init이므로 대시보드 저장에 문제가 있으면 해당 인덱스 생성 여부를 확인해 보기 바랍니다.

5.4 정리

이번 장에서 다룬 ELK는 시간 흐름에 따른 로그 데이터를 모니터링하거나 색인 데이터를 쉽게 분석할 수 있도록 제공하는 유용한 소프트웨어 스택입니다. 데이터에 대한 시각화 도구 또는 모니터링 도구가 필요하다면 ELK 구성을 추천합니다.

SQL 활용하기

SQL은 수많은 데이터베이스 관련 프로그램에서 표준으로 채택하고 있는 언어라서 SQL에 익숙한 사용자가 많을 수밖에 없습니다. 그렇다면 하둡과 같은 빅데이터 시스템에서는 SQL이 얼마나 많이 사용되고 있을까요? SQL on Hadoop 관련 엔진이 상당수 있는데, Apache Tajo, Hive, Impala 등이 대표적입니다. SQL은 백엔드 시스템에 대한 이해 수준을 낮추고 사용하는 데 편리성을 제공하는 아주 유용한 도구입니다.

그렇다면 Elasticsearch는 어떨까요? Elasticsearch는 RDBMS와 매우 유사한데, 다음 표로 간단히 비교해 보겠습니다.

표 6-1 Elasticsearch vs. RDBMS

Elasticsearch	RDBMS
Index	Database
Type	Table
Document	Row
Field	Column
Analyzer	Index
_id	Primary key
Mapping	Schema
Shard	Physical partition
Route	Logical partition
Parent-Child, Nested	Relation
Query DSL	SQL

표에서 보듯이 Elasticsearch에서는 SQL과 비슷한 Query DSL이라는 질의 언어를 제공합니다. 하지만 루씬이나 Elasticsearch 등의 검색 라이브러리와 서버에 대한 지식이나 이해가 없이 Query DSL를 처음 접하면 매우 어렵게 느껴질 수 있습니다. 그래서 이번에는 Elasticsearch를 처음 사용하거나 검색에 대한 지식이 없는 사용자도 Elasticsearch를 쉽게 사용할 수 있도록 Elasticsearch에서 SQL을 활용하는 방법을 살펴보겠습니다.

6.1 RDB 관점의 Elasticsearch

[표 6-1]에서 봤듯이 Elasticsearch와 RDB와는 유사한 점이 많습니다. 여기서는 각 항목을 좀 더 자세히 알아보겠습니다.

6.1.1 Index vs. Database

인덱스는 데이터베이스와 마찬가지로 데이터를 저장하기 위한 저장소입니다. 그러나 데이터베이스와 다른 다음 특징이 있습니다.

- 하나의 인덱스는 여러 개의 타입을 포함할 수 있습니다.
- 인덱스에 대한 alias를 이용해 서비스 중지(Downtime) 없이 인덱스 간 스위칭(Switching)이 가능합니다.

6.1.2 Type vs. Table

같은 목적 또는 구조를 가진 데이터를 저장하기 위한 저장소로 사용된다는 점이 비슷하지만 테이블과는 다른 다음 특징이 있습니다.

- _type이라는 필드에 저장되며 물리적인 저장소나 구조를 가지지 않습니다.
- _type이라는 필드를 대상으로 질의할 수 있습니다.
- 기본으로 index: "not_analyzed" 속성을 가지며 저장되지 않습니다.

6.1.3 Document vs. Row

한 개의 저장된 데이터 기준이 되는 셋^{set}을 Elasticsearch에서는 도큐먼트 document라고 부르고 RDB에서는 row 또는 record라고 합니다. 하지만 도큐먼트는 전체 필드 셋의 값^{value}을 포함하지만 row는 질의 시점의 필드 목록에 대한 값 value으로 구성됩니다. 도큐먼트의 특징은 다음과 같습니다.

- 물리적으로 인덱스 속에 존재합니다.
- 실제로 인덱스 내부의 탑재에 의해 색인 및 할당됩니다.

6.1.4 Field vs. Column

모든 테이블과 탑재은 필드라는 작은 엔티티^{Entity}로 나눠져 있는데, column은 테이블에서의 필드를 가리킵니다. 그리고 column은 테이블의 특정 필드와 관련된 모든 정보를 포함하는 엔티티이기도 합니다. 필드의 특징은 다음과 같습니다.

- _FIELD 같이 _(under bar)로 시작하는 내장 필드와 사용자 정의 필드로 구성됩니다.

6.1.5 Analyzer vs. Index

Analyzer와 인덱스는 둘 다 질의 시 조건에 해당하는 필드를 빠르게 탐색하기 위한 용도로 사용합니다. Analyzer는 다음 특징이 있습니다.

- 색인과 검색 시 각각 analyzer 구성이 가능합니다.
- 각 필드에 대한 analyzer 구성이 가능합니다.
- 필드 및 분석 특성에 따른 다양한 analyzer를 지원합니다.
- 풀 텍스트(Full text) 검색을 위한 analyzed 속성, exact match를 위한 not_analyzed 속성, 아무런 분석도 하지 않는 no 속성이 있습니다.

6.1.6 _id vs Primary Key

문서의 유일한 식별자로 사용이 된다는 점에서 비슷합니다. _id는 다음과 같은 특

징이 있습니다.

- 특정 필드 값을 지정할 수 있습니다.
- _id 값을 이용해 저장될 샴드를 선택합니다.

6.1.7 Mapping vs. Schema

문서에 대한 저장과 질의 시 필드 속성을 정의하고 각각 테이블과 타입에 대한 설정이라는 점에서 비슷합니다. 매핑의 특징은 다음과 같습니다.

- 필드 구성을 별도로 하지 않아도 동적 매핑에 의해 자동으로 데이터 타입과 문서가 저장됩니다.
- 동적 템플릿을 통해 개별 타입에 대한 유연한 관리가 가능합니다.

6.1.8 Shard/Route vs. Partition

물리적 또는 논리적으로 데이터에 대한 분산 처리 기능을 제공하고 데이터에 대한 저장과 질의 시 활용할 수 있다는 점이 비슷합니다.

샴드는 다음과 같은 특징이 있습니다.

- 데이터의 기준이 되는 기본 샴드 Primary shard 와 장애 대응 및 검색 Throughput을 높이기 위한 레플리카 샴드 Replica Shard 가 있습니다.
- 기본 샴드 크기는 한 번 정의하면 수정할 수 없지만 레플리카 샴드 크기는 동적으로 수정 할 수 있습니다.

라우트는 다음과 같은 특징이 있습니다.

- 라우트 값을 이용해 문서를 분류할 수 있습니다.
- 같은 라우트 값을 가진 문서들은 같은 샴드로 저장됩니다.

6.1.9 Parent–Child/Nested vs. Relation

데이터에 대한 정규화 과정을 통해 데이터 중복을 제거하고 관계형 구조를 생성

한다는 점에서 RDB의 Relation과 비슷합니다. 하지만 Elasticsearch에서는 1:N 관계의 최소 조건 관계형 구조와 nested 같은 변형된 관계형 구조를 제공합니다.

parent-child의 특징은 다음과 같습니다.

- 이 타입은 모두 같은 인덱스에 생성되어야 합니다.
- parent-child에 대한 질의 시 결과는 각각의 질의 결과값을 가진 문서가 나옵니다.

nested의 특징은 다음과 같습니다.

- nested path를 갖는 질의를 수행합니다.
- 비정규화된 데이터 구조를 갖기 때문에 질의 결과에 모든 문서 내용이 포함됩니다.

6.1.10 Query DSL vs. SQL

이 둘은 각각의 데이터베이스로 질의하기 위한 구조화된 언어라는 점은 비슷하지만 Query DSL은 SQL과 달리 JSON 기반으로 질의 정의를 작성합니다.

6.2 SQL 정의하기

Elasticsearch에서 지원하는 SQL에는 어떤 것이 있고, Query DSL에 대응하는 SQL문 구조 정의는 어떤지 알아보겠습니다.

6.2.1 SQL 정의

기본적인 SQL 종류와 Elasticsearch에서 제공하는 정보를 살펴보겠습니다.

DDL Data Definition Language, 데이터 정의 언어

데이터 관계를 정의하는 언어로, Elasticsearch에서는 create index, create mappings 등의 API를 지원합니다.

DML Data Manipulation Language, 데이터 조작 언어

데이터를 검색, 저장, 수정, 삭제하는 데 사용하는 명령어 조합으로, Elasticsearch에서는 select, insert, delete, upsert 명령을 지원합니다. 검색엔진 특성상 내부적으로 update는 delete 다음 insert 순서로 동작하므로 upsert로 정의합니다.

SQL 구문

검색엔진 특성상 모든 SQL 구문을 지원할 수 없습니다. 지원 가능한 내용은 [표 6-2]를 참고하기 바랍니다. 다음 표의 구문은 Apache Tajo SQL Analyzer를 이용하므로 Tajo SQL 구문을 참고하였습니다.

표 6-2 Elasticsearch에서 지원하는 SQL 구문

SQL	표현식
SELECT Statement	SELECT field1, field2, ..., fieldN FROM type_name1,type_name2,⋯,type_nameN
WHERE Clause	SELECT field1, field2, ..., fieldN FROM type_name WHERE query_condition
AND/OR Clause	SELECT field1, field2, ..., fieldN FROM type_name WHERE query_condition1 {AND OR} query_condition2
IN Clause	SELECT field1, field2, ..., fieldN FROM type_name WHERE field_name IN (value1, value2,⋯,valueN)
BETWEEN Clause	SELECT field1, field2, ..., fieldN FROM type_name WHERE field_name BETWEEN value1 AND value2
LIKE Clause	SELECT field1, field2, ..., fieldN FROM type_name WHERE field_name LIKE 'value*'

SQL	표현식
ORDER BY Clause	<pre>SELECT field1, field2, ..., fieldN FROM type_name WHERE query_condition ORDER BY field_name {ASC DESC}</pre>
GROUP BY Clause	<pre>SELECT SUM(field_name) AS RESULT_VARIABLE_NAME FROM type_name WHERE query_condition GROUP BY field_name</pre>
COUNT Clause	<pre>SELECT COUNT(field_name) FROM type_name WHERE query_condition</pre>
CREATE TABLE Statement	<pre>CREATE TABLE type_name (field1 field_type, ...) USING index_name WITH (key1=value1, ...) PARTITION BY COLUMN (field1 field_type, ...)</pre>
DROP TABLE Statement	DROP TABLE type_name
DESC Statement	DESC type_name
ALTER TABLE Statement	<pre>ALTER TABLE type_name ADD COLUMN field field_type</pre>
INSERT INTO Statement	<pre>INSERT INTO LOCATION type_name USING index_name WITH (key=value, ...)</pre>
UPSERT INTO Statement	<pre>UPSERT INTO LOCATION type_name USING index_name WITH (key=value, ...)</pre>
DELETE Statement	<pre>DELETE FROM type_name WHERE query_condition</pre>
CREATE DATABASE Statement	CREATE DATABASE index_name
DROP DATABASE Statement	DROP DATABASE index_name
USE Statement	USE index_name

SQL 데이터 타입

Elasticsearch에서 제공하는 데이터 타입은 [표 6-3]과 같습니다.

표 6-3 Elasticsearch에서 지원하는 데이터 타입

데이터 타입	설명
string	Text(도큐먼트 당 기본값16MB)
integer	-2,147,483,648~2,147,483,647
long	-9,223,372,036,854,775,808~9,223,372,036,854,775,807
float	single-precision 32-bit IEEE 754 floating point
double	double-precision 64-bit IEEE 754 floating point
boolean	true or false
date	long (internally)
array	개별 필드나 object의 리스트형
object	key:value의 Map object형
nested	Map object형
ip	ipv4
geo point	latitude, longitude
geo shape	point, linestring, polygon, multipoint, multilinestring, multipolygon, envelope, circle
attachment	encoded as base64

SQL 연산자

질의 조건에 사용한 연산자는 [표 6-4]와 [표 6-5]를 참고하기 바랍니다.

표 6-4 비교 연산자

연산자	설명	예	타입
=	문서 내 필드에서 추출된 색인어가 정확히 일치하는 경우 true	field = 'term'	string, number, date, boolean
!=	문서 내 필드에서 추출된 색인어가 일치하지 않는 경우 true	field != 'term'	

연산자	설명	예	타입
>	문서 내 필드 값이 오른쪽 값보다 클 경우 true	field > '20140623121212'	
<	문서 내 필드 값이 오른쪽 값보다 작을 경우 true	field < 2000000	number,
=	문서 내 필드 값이 오른쪽 값보다 크거나 같 을 경우 true	field > '20140623121212'	date
=	문서 내 필드 값이 오른쪽 값보다 작거나 같 을 경우 true	field < 2000000	

표 6-5 논리 연산자

연산자	설명	Example
AND	복수 개의 조건이 만족할 경우 true	field1 = 'term1' AND field2 = 'term2'
OR	복수 개의 조건 중 하나라도 만족할 경우 true	field1 = 'term1' OR field2 = 'term2'
IN	문서 내 필드 값을 목록으로 전달하고 값을 비 교하여 만족할 경우 true	field IN ('term1', ..., 'termN')
NOT	IN 연산에 대한 부정 연산자	field NOT IN ('term1', ..., 'termN')

6.2.2 인덱스 생성/삭제/선택

Elasticsearch에서 인덱스를 생성하는 명령입니다.

```
CREATE DATABASE [IF NOT EXISTS] <index_name>
```

Elasticsearch에서 인덱스를 삭제하는 명령입니다.

```
DROP DATABASE [IF EXISTS] <index_name>
```

Elasticsearch에서 질의 대상으로 사용할 인덱스를 선택하는 명령입니다.

```
USE <index_name>
```

6.2.3 테이블 생성/삭제

Elasticsearch에서 인덱스 내 도큐먼트 타입을 생성하는 명령입니다.

```
CREATE TABLE <type_name> (
    [(<field_name> <field_type>, …)]
)
USING index_name
WITH (
    [(<key>=<value>, …)]
)
PARTITION BY COLUMN (
    [(<field_name> <field_type>, …)]
);

```

다음 옵션은 Elasticsearch에서 매팅 설정 시 지정한 타입 내에서 global 설정이 됩니다.

표 6-6 옵션

KEY	VALUE
'_parent'	'parent_type'
'_id'	'document_id_field'
'_all'	'disable'
'_source'	'enable'
'_routing'	'routing_field'
'analyzer'	'analyzer_name'

Elasticsearch에서 인덱스 내 도큐먼트 타입을 삭제하는 명령입니다.

```
DROP TABLE [IF EXISTS] <type_name>
```

6.2.4 절 선택/삽입/업데이트/삭제

Elasticsearch의 Query DSL에서 검색 질의에 해당하는 명령입니다.

```
SELECT  
  [<field_name>, …]  
  [FROM <type_name>, …]  
  [WHERE <condifion>, …]  
  [GROUP BY <field_name>, …]  
  [ORDER BY (<field_name> [ASC|DESC]), …]  
  [LIMIT <offset>/<row_count>]
```

Elasticsearch에서 문서 색인에 해당하는 명령입니다.

```
INSERT INTO LOCATION <type_name>  
USING <index_name>  
WITH (  
  [(<field_name>=<value>× …)]  
)
```

Elasticsearch 내부적으로 도큐먼트의 update 작업은 _id 값을 기준으로 생성 후 삽입하는 순서로 동작하므로 upsert로 정의합니다.

```
UPSERT INTO LOCATION <type_name>  
USING <index_name>  
WITH (  
  [(<field_name>=<value>× …)]  
)  
SELECT <ID>
```

Elasticsearch에서 문서를 삭제하는 명령입니다. 질의 조건에 만족하는 문서를 삭제하고, 삭제 후 실시간으로 검색 조건에 반영하기 위해서는 refresh 명령어를 실행해야 합니다.

```
DELETE FROM <type_name> WHERE [conditions]
```

6.3 SQL 변환하기

앞에서 정의한 SQL을 기반으로 SQL이 Query DSL로 어떻게 매핑되고 변환되는지 살펴보겠습니다. 이를 통해 JDBC 드라이버를 만들거나 SQL문이 검색엔진에서 변환되어 동작하는 과정을 쉽게 이해할 수 있습니다. 하지만 모든 SQL문이 Query DSL로 변환되는 것은 아니므로 이 점을 유의하기 바랍니다.

6.3.1 Match_all Query

match_all 쿼리는 조건절 없이 모든 데이터를 가져오는 데 사용하는 질의로 SQL문의 select all과 같은 의미입니다.

Query DSL

match_all 쿼리는 다음과 같이 다양하게 작성할 수 있습니다.

```
"match_all": {}

"bool": {
    "must": [
        {
            "match_all": {}
        }
    ]
}

"query_string": {
    "query": "*"
}
```

SQL

질의문에서 보듯이 where 조건절이 없습니다.

```
SELECT *
FROM type_name
```

6.3.2 Match Query

match 쿼리는 질의어 자체를 분석하여 색인어를 추출하고 검색을 수행합니다. 이 때문에 불필요한 분석 과정이 추가되므로 match 쿼리보다는 term 쿼리 사용을 추천합니다.

Query DSL

기본으로 value에 대한 분석 작업을 합니다.

```
"match": {  
    "field": "value"  
}
```

SQL

```
SELECT *  
FROM type_name  
WHERE field = value
```

6.3.3 Bool Query

bool 쿼리는 조건을 만족하는 도큐먼트를 가져오는 쿼리로, 이 쿼리 하나로 서로 다른 쿼리 타입을 조합하여 사용할 수 있어서 Elasticsearch에서 사용하는 쿼리 중 유용하고 편리한 쿼리의 하나입니다. 이 쿼리는 조건에 대한 다음 표의 타입이 있습니다.

표 6-7 Bool 타입

타입	설명
must	<ul style="list-style-type: none">이 타입으로 등록된 쿼리는 반드시 조건을 만족해야 합니다.SQL문의 EQUAL과 AND에 해당합니다.

타입	설명
should	<ul style="list-style-type: none"> 이 타입으로 등록된 쿼리는 조건 중 만족하는 내용이 하나만 있어도 됩니다. SQL문의 IN과 OR에 해당합니다.
must_not	<ul style="list-style-type: none"> 이 유형으로 등록된 쿼리는 반드시 조건과 불일치해야 합니다. SQL문의 NOT에 해당합니다.

Query DSL

각 필드의 조건을 모두 만족하는 도큐먼트를 찾습니다.

[bool 쿼리 - must]

```
"bool": {
  "must": [
    {
      "match": {
        "field1": "value1"
      }
    },
    {
      "match": {
        "field2": "value2"
      }
    }
  ]
}
```

각 필드의 조건 중 하나 이상 일치하는 문서를 찾습니다. 두 필드가 서로 다른 필드가 아니고 같은 필드라면 terms 쿼리를 사용하는 것이 좋습니다.

[bool 쿼리 - should]

```
"bool": {
  "should": [
    {
      "match": {
        "field1": "value1"
      }
    }
  ],
  "must": [
    {
      "match": {
        "field2": "value2"
      }
    }
  ]
}
```

```
{  
    "match": {  
        "field2": "value2"  
    }  
}  
]  
}
```

지정한 필드의 value와 같은 값이 없는 문서를 찾습니다.

[bool 쿼리 - must_not]

```
"bool": {  
    "must_not": [  
        {  
            "match": {  
                "field": "value"  
            }  
        }  
    ]  
}
```

이 조합 쿼리는 AND, OR 등의 연산자 우선순위에 따라 작성해야 합니다.

[bool 쿼리 - 조합]

```
"bool": {  
    "must": [  
        {  
            "match": {  
                "field1": "value1"  
            }  
        },  
        {  
            "bool": {  
                "should": [  
                    {  
                        "match": {  
                            "field2": "value2"  
                        }  
                    },  
                    {  
                        "match": {  
                            "field3": "value3"  
                        }  
                    }  
                ]  
            }  
        }  
    ]  
}
```

```
{  
    "match": {  
        "field3": "value3"  
    }  
}  
]  
}  
]  
}
```

SQL

[AND]

```
SELECT *  
FROM type_name  
WHERE field1 = value1  
AND field2 = value2
```

[OR]

```
SELECT *  
FROM type_name  
WHERE field1 = value1  
OR field2 = value2  
[NOT]  
SELECT *  
FROM type_name  
WHERE field != value
```

[조합]

```
SELECT *  
FROM type_name  
WHERE field1 = value1  
AND (  
    field2 = value2  
    OR field3 = value3  
)
```

bool 쿼리에서 가장 큰 장점은 다양한 쿼리를 조합해서 사용할 수 있다는 점입니다. 앞의 예제에서는 주로 match 쿼리만 사용되었지만 ids, range, term 쿼리 등을 모두 사용할 수 있고, 이를 다양한 조건식에 활용할 수 있습니다.

6.3.4 Ids Query

이 쿼리는 전형적인 SQL의 IN절 쿼리입니다. 질의 시 도큐먼트의 _id 필드와 값이 일치하는 문서를 찾습니다.

Query DSL

이 쿼리는 _id 필드에 대해서만 조건 검색을 합니다.

```
"ids": {  
    "values": [ value1, value2, value10 ]  
}
```

SQL

```
SELECT *  
FROM type_name  
WHERE _id IN (value1, value2, value10)
```

6.3.5 Range Query

숫자나 날짜 필드에 대한 범위 검색을 하는 쿼리로, SQL문의 부등호 연산자를 포함한 조건식과 BETWEEN/AND 쿼리와 비슷합니다. 사용하는 연산자의 의미는 다음 표와 같습니다.

표 6-8 Range 쿼리 연산자

연산자	설명
from	필드의 왼쪽 value에 해당하며, 값을 포함합니다.
to	필드의 오른쪽 value에 해당하며, 값을 포함합니다.
gte	\geq (Greater than equal)
gt	$>$ (Greater than)
lte	\leq (Less than equal)
lt	$<$ (Less than)

Query DSL

다음 두 쿼리는 의미가 같습니다.

```
"range": {  
    "field": {  
        "from": left_value,  
        "to": right_value  
    }  
}  
//////////  
  
"range": {  
    "field": {  
        "gte": left_value,  
        "lte": right_value  
    }  
}
```

SQL

```
SELECT *  
FROM type_name  
WHERE field >= left_value  
AND field <= right_value  
//////////  
SELECT *
```

```
FROM type_name  
WHERE field  
BETWEEN left_value AND right_value
```

6.3.6 Term Query

term 쿼리는 지정한 필드에서 색인어가 일치하는 문서를 찾는 점은 match 쿼리와 비슷하지만 match 쿼리와 다르게 질의 조건으로 할당된 value에 대한 분석 작업을 하지 않고 value 자체가 색인어가 됩니다.

예를 들어, "field = value"에서 match 쿼리는 value를 지정된 analyzer를 통해 색인어를 만들고, term 쿼리는 value가 바로 색인어가 됩니다.

Query DSL

value에 대한 분석 과정 없이 색인어로 사용됩니다.

```
"term": {  
    "field": "value"  
}
```

SQL

```
SELECT *  
FROM type_name  
WHERE field = value
```

6.3.7 Terms Query

terms 쿼리는 한 필드에서 여러 개의 색인어를 검색하고 싶을 때 사용합니다. match 쿼리와 term 쿼리의 장점을 섞어 놓은 쿼리로, SQL문의 IN절과 비슷합니다.

Query DSL

필드에서 value1 또는 value2와 일치하는 문서를 찾습니다.

```
"terms": {  
    "field": [ "value1", "value2" ]  
}
```

SQL

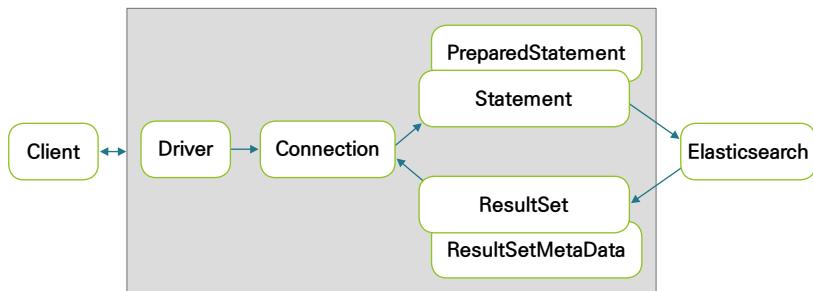
```
SELECT *  
FROM type_name  
WHERE field IN ( value1, value2 )
```

6.4 JDBC Driver 만들기

앞의 과정은 모두 JDBC 드라이버를 만들기 위해 Query DSL을 SQL로 어떻게 구성할지 살펴본 것입니다. 이를 바탕으로 JDBC 드라이버를 구현하는 과정을 살펴보겠습니다.

다음 그림은 Elasticsearch용 JDBC 드라이버를 만드는 데 필요한 클래스 간 개념도입니다.

그림 6-1 JDBC 드라이버 개념도



각 클래스를 상속받아 Elasticsearch용 JDBC 드라이버를 만듭니다. 각 클래스를 자세히 살펴보겠습니다.

6.4.1 Elasticsearch Driver

JDBC 드라이버에서 가장 기본이 되는 클래스로, 클라이언트의 프로그램에서 JDBC 드라이버를 사용할 수 있게 드라이버를 등록하고 Connection 객체를 생성하는 역할을 합니다.

다음은 ElasticsearchDriver의 샘플 코드입니다.

[ElasticsearchDriver.java]

```
public class ElasticsearchDriver implements Driver, Closeable {
    ...중략...
    static {
        try {
            DriverManager.registerDriver(new ElasticsearchDriver());
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    ...중략...
    @SuppressWarnings("resource")
    @Override
    public Connection connect(String url, Properties properties)
        throws SQLException {
        url = url.replaceAll(";", "&");
        return acceptsURL(url) ? new ElasticsearchConnection(url, properties) : null;
    }
    ...중략...
}
```

6.4.2 Elasticsearch Connection

Connection 객체는 기본으로 연결 URI에 대한 유효성 검사와 Elasticsearch

용 Statement 객체를 생성하고 Connection 인터페이스에 대한 메서드를 구현해야합니다.

다음은 ElasticsearchConnection의 샘플 코드입니다.

[ElasticsearchConnection.java]

```
public class ElasticsearchConnection implements Connection {  
    ...중략...  
    public ElasticsearchConnection(String rawURI, Properties properties) throws  
    SQLException {  
        ...중략...  
    }  
    ...중략...  
}
```

6.4.3 Elasticsearch Statement

SQL문의 처리와 실행을 담당하고 작성된 SQL문을 Query DSL로 변환하여 Elasticsearch로 질의합니다. 기본으로 Statement 인터페이스에 대한 메서드를 구현해야 합니다.

다음은 ElasticsearchStatement의 샘플 코드입니다.

[ElasticsearchStatement.java]

```
public class ElasticsearchStatement implements Statement {  
    ...중략...  
    public ElasticsearchStatement(ElasticsearchClient esClient) {  
        ...중략...  
    }  
  
    public Client getClient() throws SQLException {  
        return esClient.getClient();  
    }  
  
    public Settings getSettings() throws SQLException {  
        return esClient.getSettings();  
    }  
}
```

```
@Override  
public ResultSet executeQuery(String sql) throws SQLException {  
    ...중략...  
}  
  
@Override  
public int executeUpdate(String sql) throws SQLException {  
    ...중략...  
}  
}  
...중략...
```

- `executeQuery()`에서는 SELECT절을 실행합니다.
- `executeUpdate()`에서는 INSERT, UPSERT, DELETE, CREATE, DROP을 실행합니다.

6.4.4 Elasticsearch ResultSet

질의 실행에 대한 결과를 처리하고 저장하는 역할을 합니다. 특히 Elasticsearch의 결과는 일반 RDB와는 매우 달라서 Elasticsearch 전용 API를 추가하여 사용하면 매우 편리합니다. 기본으로 `ResultSet` 인터페이스에 대한 메서드를 구현해야 합니다.

다음은 `ElasticsearchResultSet`의 샘플 코드입니다.

```
[ ElasticsearchResultSet.java ]  
  
public class ElasticsearchResultSet implements ResultSet {  
    ...중략...  
    public ElasticsearchResultSet() {  
    }  
  
    public ElasticsearchResultSet(SearchResponse searchResponse,  
        IEsDataReader reader,  
        int fetchSize,  
        int offset,  
        int limit) throws SQLException {  
        ...중략...  
    }  
}
```

```

    }

    public SearchResponse getSearchResponse() {...}
    public int getFailedShards(){...}
    public int getSuccessfulShards(){...}
    public long getTookInMillis(){...}
    public int getTotalShards(){...}
    public Aggregations getAggregations(){...}
    public String getSearchResponse2String(){...}
    public long getTotalSize() throws SQLException {...}

    @Override
    public String getString(int columnIndex) throws SQLException {...}
    ...중략...
}

```

- Elasticsearch에 대한 특화된 메서드를 만들어야 합니다.
- Elasticsearch의 데이터 타입을 처리해야 합니다.
- 각 데이터 타입에 따른 컬럼 인덱스나 컬럼 라벨^{column label}로 데이터를 읽을수 있는 GET 메서드를 만들어야 합니다.
- 메타 데이터 정보를 구성하기 위한 GET/SET 메서드를 만들어야 합니다.

6.4.5 Elasticsearch ResultSetMetaData

Elasticsearch의 타입의 매핑 정보를 저장하는 역할을 하며, ResultSet에서 관련 정보가 설정됩니다. 기본으로 ResultSetMetaData 인터페이스에 대한 메서드를 구현해야 합니다.

다음은 ElasticsearchResultSetMetaData의 샘플 코드입니다.

```
[ ElasticsearchResultSetMetaData.java ]
public class ElasticsearchResultSetMetaData implements ResultSetMetaData {
    private List<String> columnNames;
    private List<String> columnTypes;
    private int columnCount;

    ElasticsearchResultSetMetaData(List<String> columnNames, List<String> columnTypes, int
        columnCount) {
```

```
    ...중략...
}

@Override
public String getColumnName(int column) throws SQLException {
    return columnNames.get(column - 1);
}

@Override
public int getColumnType(int column) throws SQLException {
    return types.get(columnTypes.get(column-1));
}
...중략...
}
```

6.4.6 Elasticsearch JDBC Driver 예제

다음은 실제로 구현한 Elasticsearch JDBC 드라이버를 이용하여 테스트한 예제 코드입니다.

```
public void testElasticsearchJDBCDriver() throws Exception {
    Connection conn = null;
    EsStatement stmt = null;

    try {
        Class.forName("com.gruter.elasticsearch.esbase.jdbc.EsDriver");
        conn = DriverManager.getConnection("jdbc:es://localhost:9300/elasticsearch?data
base=devview2014&shards=12&.....", "", "");
        stmt = (EsStatement) conn.createStatement();

        String sql = "SELECT * FROM elasticsearch WHERE price > 1050 ORDER BY price DESC
LIMIT 0/10";
        EsResultSet rs = (EsResultSet) stmt.executeQuery(sql);

        log.debug("{}" , rs.getSearchResponse());

        rs.close();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        ...중략... // close 처리
    }
}
```

```
    }  
}
```

6.5 정리

Elasticsearch에서의 SQL를 활용하려면 기본으로 JDBC 드라이버를 구현해야 합니다. 여기서는 전체 구현 코드를 보여주지 못했지만 구현하는 방법은 충분히 설명하였습니다. 이미 구현된 JDBC 드라이버 소스 코드를 참고하여 구현할 수 있으므로 github와 같은 곳에서 관련 코드를 참고하시면 좋을 것 같습니다. 검색 엔진에 대한 이해와 지식이 없는 사용자에게 SQL이라는 도구는 정말 유용하게 사용될 수 있다는 점을 이해하시면 좋을 것 같습니다.

Elasticsearch 성능 최적화

Elasticsearch의 성능 최적화는 정답이 정해져 있지 않으며 상황에 맞춰 다양한 방법을 적용해 보고 테스트해 봐야 합니다. 가장 쉽게는 하드웨어 scale up/out을 통해 순간적인 성능 향상을 얻을 수 있지만 이것은 임시방편일 뿐 근본적인 해결책이 되지 못합니다. 여기서는 성능 향상을 위한 접근 방법과 성능 향상을 시도 할 수 있는 항목을 살펴보고 어떻게 테스트하면 되는지를 알아보겠습니다.

Elasticsearch에서 성능 향상이 필요한 기능은 검색과 색인입니다. 이 기능은 장비 스펙, 네트워크 자원, 색인 문서 특성, 질의 특성 등에 따라 성능 튜닝 포인트가 달라집니다. 성능 향상을 위한 기본 방법은 다음과 같습니다.

- Step 1) 실제 운영환경과 같은 스펙의 장비를 같은 위치에 하나 더 구성합니다.
- Step 2) 실제 운영환경과 같은 설정의 인덱스를 생성합니다. 단, 복제 기능은 사용하지 않으며 하나의 인덱스에 하나의 샤프만 생성합니다.
- Step 3) 실제 운영환경에서 사용하는 동일한 문서를 대상으로 색인 작업을 합니다.
- Step 4) 구성된 운영 데이터를 기반으로 질의와 aggregation을 수행합니다.

3-4번 과정에서 해당 서버의 성능 수치를 구할 수 있으므로 샤프 구성과 노드 구성은 어떻게 할 것인지 정의할 수 있는 근거를 얻을 수 있습니다.

7.1 하드웨어 관점

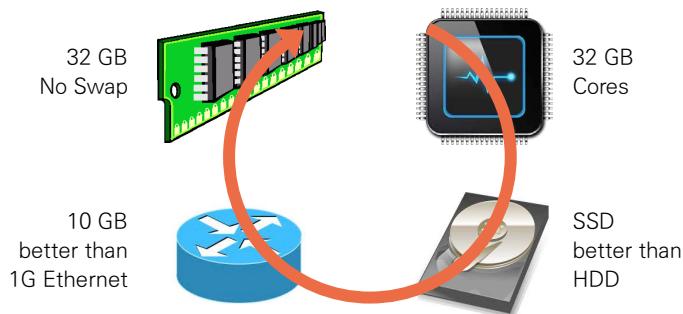
운영환경과 동일한 장비를 구성하여 테스트하는 것이 매우 중요합니다. 기본으로

CPU, 램, 디스크, 네트워크 등의 정보를 확인해야 합니다. Elasticsearch의 장비 리소스에 대한 최대 사용값은 CPU가 32코어, 램이 32GB로 코드에 정의되어 있습니다. 그러므로 무조건 CPU와 램을 증설한다고 해서 성능이 좋아지지는 않습니다.

하드웨어 관점에서 성능을 튜닝하려면 색인과 검색 시 iostat, netstat, vmstat 등으로 병목 구간을 확인해야 합니다.

그림과 같이 성능 문제는 장비 스펙 및 영향도에 따라 분석하는 것이 좋습니다.

그림 7-1 하드웨어 관점의 성능 튜닝



7.1.1 CPU

하이퍼 스레딩을 포함해 32코어까지 사용할 수 있습니다. 32코어보다 크면 너무 많은 스레드가 생성되어 OOM^{Out Of Memory}이 발생할 수 있어서 이를 방지하기 위해 제한합니다.

CPU 코어 값은 Elasticsearch에서 모든 기능 수행에 필요한 스레드 풀의 크기를 정의하는 기준이 됩니다. 보통은 기본값을 사용해도 문제가 되지 않지만 색인과 검색에 대한 질의 특성과 Elasticsearch의 사용 용도에 따라 스레드 풀의 크기를 조정해 사용합니다.

다음 표는 스레드 풀 Thread Pool 항목입니다.

표 7-1 스레드 풀 항목

항목	설명
bulk	Bulk 연산에 대한 스레드 풀
flush	Flush 연산에 대한 스레드 풀
generic	Generic 연산에 대한 스레드 풀(보통 클러스터와 노드에 대한 연산에서 사용)
get	Get 연산에 대한 스레드 풀
index	Index 연산에 대한 스레드 풀
management	Management 연산에 대한 스레드 풀(주로 리소스 모니터링에 대한 연산에 사용)
merge	Merge 연산에 대한 스레드 풀
optimize	Optimize 연산에 대한 스레드 풀
percolate	Percolator 연산에 대한 스레드 풀
refresh	Refresh 연산에 대한 스레드 풀
Search	Search/count 연산에 대한 스레드 풀
snapshot	Snapshot 연산에 대한 스레드 풀
suggest	Suggest 연산에 대한 스레드 풀
warmer	Index warm-up 연산에 대한 스레드 풀

다음 표는 스레드 풀 설정 시 사용하는 필드 항목입니다.

표 7-2 스레드 풀의 필드 항목

필드	설명
type	스레드 풀의 타입(cached, fixed, scaling)
active	Active 스레드 풀의 크기
size	현재 설정된 스레드 풀의 크기
queue	현재 설정된 스레드 풀의 큐 task 크기
queueSize	현재 설정된 스레드 풀의 큐 task 최대 크기
rejected	현재 스레드 풀에서 거절된 스레드 수
largest	현재 스레드 풀에서 실행된 최대 스레드 수
completed	현재 스레드 풀에서 task를 완료한 스레드 수
min	현재 설정된 스레드 풀의 최소 크기
max	현재 설정된 스레드 풀의 최대 크기
keepAlive	현재 설정된 스레드 풀의 keep alive 시간

CPU는 검색과 색인에 대한 처리를 할 때 가장 많은 리소스를 사용합니다. 하지만 Elasticsearch의 노드 특성상 마스터나 클라이언트 노드는 CPU 리소스를 상대적으로 적게 사용합니다. 이런 특성을 이용하여 노드 구성을 다양하게 시도해 보면 안정성과 성능 향상에 도움이 됩니다.

CPU 코어와 더불어 CPU 사용률^{Utilization}도 매우 중요합니다. CPU 사용률이 너무 높으면 처리율이 떨어지고 몇몇 연산은 실패할 확률이 높아지게 됩니다. 이것은 근본적으로 안정성과 성능 문제를 유발하므로 CPU 사용률이 꾸준하게 60%를 넘으면 scale out을 고려해 보는 것이 좋습니다.

7.1.2 RAM

램은 최대 32GB까지 설정할 수 있습니다. 램 용량을 더 크게 설정하면 GC Garbage Collection 수행 시 성능 저하를 유발할 수 있으므로 노드 구성에 따라 적절하게 설정하는 것이 좋습니다.

램에 상주하는 데이터는 쿼리 캐시, 필터, 비트셋, 필드 데이터 등 매우 다양한 종류가 있습니다. 또한, 검색 질의 중 분석을 위한 aggregation 쿼리 실행할 때 대상 데이터가 너무 크면 성능이 저하되거나 검색엔진이 다운되는 현상도 발생할 수 있으니 주의해야 합니다.

가상 메모리

리눅스 계열 운영체제에서는 mmap count 제한으로 OOM이나 성능 저하가 발생할 수 있으므로 다음 명령어로 mmap count를 늘려줍니다.

```
sysctl -w vm.max_map_count=262144
```

메모리 설정

리눅스 커널은 파일 시스템을 위해 가능한 한 많은 메모리를 사용하려고 하므로

결과적으로 Elasticsearch에서 swap을 사용하는 원인이 되기도 합니다. swap을 사용하면 안정성과 성능 측면에서 매우 나쁜 결과가 나올 수 있으므로 다음과 같은 설정으로 방지해야 합니다.

- **Disable swap** : swap을 방지하기 위한 가장 쉬운 방법으로, Elasticsearch 전용 장비에 이 설정을 적용하면 성능면에서 효과적입니다.
- **Configure swapping** : 다른 방법으로는 sysctl의 vm.swappiness 값을 0으로 설정하는 것입니다. 이 설정으로 swap을 사용하게 되는 상황을 줄이거나 피할 수 있습니다.
- **mlockall** : 이 설정은 Linux/Unix 시스템에서만 한정적으로 사용할 수 있습니다. 이는 Elasticsearch의 전용 메모리 공간에 락 설정을 해서 swap 사용을 방지하는 방법입니다. 이 설정을 사용하려면 ulimit -l unlimited 설정을 먼저 한 후 Elasticsearch를 실행해야 합니다.

7.1.3 DISK

Elasticsearch에서는 기본으로 SSD 사용을 권장합니다. 루씬을 사용하는 검색 엔진 특성상 세그먼트 파일에 대한 병합^{Merge} 작업 수행 시 성능 저하 현상이 발생 하므로 싱글 스레드로 동작하는 spinning 디스크보다는 멀티 스레드로 동작하는 SSD가 더 적절합니다. 하지만 SSD 사용이 필수 조건은 아니므로 상황에 맞춰 사용하면 됩니다.

Elasticsearch에서 디스크는 기본적인 read/write에 대한 처리율 때문에 중요 합니다. 처리율은 디스크 스펙뿐만 아니라 CPU와 네트워크 스펙과도 연관이 있으므로 어느 하나의 문제로만 봐서는 안 되며 복합적인 검토를 바탕으로 성능 문제를 해결해야 합니다.

7.1.4 NETWORK

Elasticsearch의 중요한 특징의 하나는 분산 구조 Distributed Architecture 지원입니다.

다. 이는 데이터가 네트워크를 통해 각각의 노드에 분산되어 저장되고 검색되는 것을 의미합니다. 즉, 모든 데이터는 네트워크를 통해서 전달됩니다. 따라서 Elasticsearch 클러스터가 구성된 네트워크의 대역폭(Bandwidth)이 어떻게 구성되어 있는지 확인하여 병목이 발생하거나 전체 대역폭이 사용되지 않도록 주의해야 합니다.

네트워크 상황이 좋지 못하면 각 노드간 ping 또는 연산에 대한 요청 시간 초과(request timeout)가 발생하여 예기치 않은 문제가 생길 수 있으니 주의하기 바랍니다.

7.2 Document 관점

Elasticsearch에서 기준이 되는 데이터는 도큐먼트입니다. 이 도큐먼트를 어떻게 정의하고 색인하는지에 따라 성능 차이를 보입니다. 여기서는 도큐먼트에 대한 모델링과 분산 배치를 통해 어떻게 성능을 개선할 수 있는지 알아보겠습니다.

7.2.1 Index와 Shard 투닝

Elasticsearch에서 도큐먼트에 대한 색인을 하려면 먼저 인덱스를 생성하고, 이 인덱스를 생성하는 데 필요한 설정값 중 기본 샤프드 와 레플리카 샤프드에 대한 크기를 정의해야 합니다. 성능 투닝 관점에서는 레플리카 샤프드는 필요하지 않으므로 disable로 처리하고 기본 샤프드만 설정하겠습니다. 기본 샤프드의 설정 투닝은 인덱스에 대한 파티션(Partition) 작업이라고 생각하면 됩니다. 샤프드 크기는 다음 기준을 참고하여 정의합니다.

데이터 노드 크기

샤프드 크기를 설정하는 가장 쉬운 판단 기준입니다. 데이터 노드가 3개라면 기본 샤프드 크기를 3개로 설정하고, 5개라면 5개로 설정하면 됩니다.

[데이터 노드가 3개일 때]

index.number_of_shards: 3

[데이터 노드가 5개일 때]

index.number_of_shards: 5

기본 샤프트의 크기는 초기에 인덱스를 생성할 때 결정되며 중간에 수정할 수 없으므로 정확한 크기 설정 근거를 기준으로 생성해야 합니다(레플리카 샤프트는 중간에 수정할 수 있습니다).

CPU 코어 크기

기본적으로 Elasticsearch는 병렬처리를 기반으로 동작합니다. 따라서 CPU 코어 크기를 샤프트 크기를 정할 때 판단 기준으로 사용할 수 있습니다. 4코어 CPU를 사용하고 있다면 데이터 노드 내 4개의 샤프트를 배치할 수 있고, 24코어 CPU를 사용하고 있다면 데이터 노드 내 24개의 샤프트를 배치할 수 있습니다.

다음은 하나의 데이터 노드에 CPU 기준으로 설정한 것과 복수 개의 데이터 노드와 연관지어 CPU 기준으로 설정한 것을 보여줍니다.

[1개의 데이터 노드 + 4코어]

index.number_of_shards: 4

[4개의 데이터 노드 + 4코어]

index.number_of_shards: 16

샤프트 크기를 늘리면 일반적인 색인과 검색에 대한 성능 향상 효과를 얻을 수 있지만 적절한 크기 이상으로 설정하면 오히려 성능 저하와 운영상의 어려움을 겪을 수 있으니 주의하기 바랍니다. 적절한 샤프트 크기는 7장 앞부분에서 설명한 Step

1부터 Step 4까지 단계를 통해서 산출해야 합니다.

도큐먼트 크기

도큐먼트 크기는 샤드 하나에 몇 개의 도큐먼트를 저장할 수 있는지에 대한 근거로 사용할 수 있습니다. 일반적으로 샤드 하나의 물리적인 크기는 50GB 이하로 정의하는 것이 좋습니다. 물론 그 이상도 저장할 수 있지만 문제 발생 시 복구와 검색 성능에 나쁜 영향을 미칠 수 있기 때문에 추천하지 않습니다.

다음은 도큐먼트 하나의 크기를 기준으로 한 크기 정의입니다. 더는 색인할 데이터가 없으면 정적 데이터를 기준으로 샤드 크기를 정의한다는 점에 유의합니다.

```
# Document 1개 크기: 100KB  
# Total document count: 10000000개  
# Total indexing 크기: 954GB  
index.number_of_shards: 20
```

7.2.2 Modeling

이번에는 도큐먼트에 대한 매핑 작업을 통한 튜닝을 살펴보겠습니다. 모델링은 색인과 검색에 대한 요구사항을 반영하여 성능 개선 효과를 보는 작업입니다.

_all 필드

이 필드의 특징은 모든 필드의 문자열에 대한 색인 작업 결과를 가지고 검색할 수 있게 지원하는 통합 필드입니다. 도큐먼트에 대한 풀 텍스트 검색 기능을 사용하는 것이 아니라면 disable로 설정하여 자원을 절약할 수 있습니다. _all 필드 사용은 타입을 정의할 때 설정하며 개별 필드 단위로도 설정할 수 있습니다.

_all enable

[탑급 내 설정]

```
"_all" : {
```

```
    "enabled" : true  
}
```

[필드 내 설정]

```
"field" : {"type" : "string", ...,"include_in_all" : true}
```

_all disable

[타입 내 설정]

```
"_all" : {  
    "enabled" : false  
}
```

[필드 내 설정]

```
"field" : {"type" : "string", ...,"include_in_all" : false}
```

_all 필드 특징

모든 필드에서 색인 작업을 수행하므로 CPU 사용량이 많고, 모든 색인 데이터를 디스크에 쓰므로 인덱스 크기도 늘어납니다.

- CPU 사용량 많음
- 디스크 쓰기 크기 증가
- 인덱스 크기 증가

_source 필드

색인 시 자동으로 생성되는 필드로, 검색 용도가 아닌 단순 화면 출력용으로만 사용합니다. 이는 매핑 설정 시 필드별 store 옵션과는 별개로 사용 여부를 지정할 수 있습니다. _all 필드와는 다르게 색인 과정이 없고 단순히 저장만 하므로 성능 면에서 큰 효과를 기대하기는 힘들지만 저장소를 절약하는 방법으로 활용할 수 있습니다.

_source enable

[타입 내 설정]

```
"_source" : {  
    "enabled" : true  
}
```

_source disable

[타입 내 설정]

```
"_source" : {  
    "enabled" : true  
}
```

_source 필드 특징

이 필드는 모든 필드를 저장하기 때문에 저장소 공간이 커질 수 있습니다.

_id 필드

도큐먼트의 기본키로 사용하는 필드로, 이 필드의 값을 기준으로 도큐먼트를 어느 색에 저장할지 결정합니다. _id 필드에 대한 특정 필드 패스를 지정하지 않으면 자동으로 랜덤한 해시^{Hash} 문자열을 가지게 됩니다.

다음은 _id 값에 따라 120개의 색에 도큐먼트가 어떻게 할당되는지 확인하기 위한 예제입니다. 지정한 필드의 value에 의해 모든 도큐먼트가 색에 잘 분포되는지 확인해야 합니다. 특정 색으로 도큐먼트가 몰리면 해당 데이터 노드에 대한 타임아웃이나 Elasticsearch 데몬에 대한 ‘응답없음’ 상태가 발생할 수 있습니다.

[색 할당 예제]

```
public class EsHashPartitionTest {  
    private static final Logger log = LoggerFactory.getLogger(EsHashPartitionTest.class);  
    private HashFunction hashFunction = new DjbHashFunction();
```

```

    @Test
    public void testHashPartition() {
        int shardSize = 120;
        List<Long> shards = new ArrayList<Long>();
        long[] partSize = new long[shardSize];

        for ( int i=0; i<shardSize; i++ ) {
            shards.add((long) 0);
            partSize[i] = 0;
        }

        for ( int i=0; i<100000; i++ ) {
            int shardId = MathUtils.mod(hash(String.valueOf(i)), shardSize);
            shards.add(shardId, (long) ++partSize[shardId]);
        }

        for ( int i=0; i<shardSize; i++ ) {
            log.debug("[{}]", partSize[i]);
        }
    }

    public int hash(String routing) {
        return hashFunction.hash(routing);
    }
}

```

샤드 할당 결과

```

DEBUG: hanb.elasticsearch.expert.func.EsHashPartitionTest - [0] 756
DEBUG: hanb.elasticsearch.expert.func.EsHashPartitionTest - [1] 755
DEBUG: hanb.elasticsearch.expert.func.EsHashPartitionTest - [2] 1001
...중략...
DEBUG: hanb.elasticsearch.expert.func.EsHashPartitionTest - [117] 754
DEBUG: hanb.elasticsearch.expert.func.EsHashPartitionTest - [118] 751
DEBUG: hanb.elasticsearch.expert.func.EsHashPartitionTest - [119] 1001

```

routing 필드

이 필드는 특정 필드 값을 이용하여 도큐먼트를 같은 샤드로 지정하여 저장하는 기능을 제공합니다. 이 기능은 검색 시 분산된 샤드로 질의를 실행하지 않고 특정

샤드에서만 질의를 수행하므로 검색 성능이 향상하는 효과를 얻을 수 있습니다. 이 필드는 기본적으로 다음 두 옵션에 대한 설정을 가집니다.

- **store:no** : 데이터를 저장하지 않습니다.
- **index:not_analyzed** : 하나의 토큰으로 색인합니다.

필드 Store 설정

하나의 도큐먼트는 여러 개의 필드로 구성되며 각각의 필드 단위로 저장 여부를 설정할 수 있습니다. 이 기능은 저장소의 용량 산정에 기준이 되므로 효율적인 저장소 관리를 위해 필요한 필드만 저장되도록 설정하는 것이 좋습니다.

필드 Analyzer 설정

색인 시 도큐먼트의 필드별로 analyzer를 설정할 수 있으며 이를 통해 다양한 분석과 질의가 가능합니다. 저장되는 값의 유형에 따라 analyzer를 설정하여 불필요한 작업을 제거하는 것이 좋습니다.

7.3 Operation 관점

Operation 관점의 튜닝은 개별 서비스에 대한 요구사항과 질의 특성에 따라 최적화를 수행해야 하는데, 모든 서비스의 특징과 질의 형태가 다르므로 여기서는 일반적인 내용을 알아보겠습니다.

7.3.1 설정 튜닝

elasticsearch.yml과 같은 설정 파일이나 실행 시 JVM 옵션 설정 등으로 튜닝하는 방법입니다.

디스크 확장

Elasticsearch는 데이터를 저장할 때 기본 로컬 저장소를 사용하므로 대용량 데

이터를 저장하려면 디스크 용량 확장이 가능해야 합니다. 용량 확장은 다음으로 설정할 수 있습니다. path.data 설정은 여러 개의 디스크를 묶어서 사용할 수 있게 해주고, 이를 이용해서 RAID 0 같은 구성과 디스크 입/출력에 대한 성능 개선 효과를 얻을 수 있습니다.

```
path.data: /disk1/data, /disk2/data
```

Node Topology

Elasticsearch는 노드에 역할을 부여할 수 있어서 목적과 용도에 맞춰 클러스터 안에 노드를 구성하여 검색과 색인에 대한 성능 개선과 안정성을 확보할 수 있습니다.

검색 성능 향상을 위한 검색 로드 밸런서 노드 구성

검색 로드 밸런서 노드 Search Load Balancer Node는 응용 프로그램에서 질의 요청을 받아 데이터 노드로 요청을 분산하고 요청에 대한 결과를 취합하여 응용 프로그램으로 응답하는 역할을 수행합니다. 따라서 이런 구성을 한 노드에는 물리적 데이터가 없고, 디스크 용량도 많이 필요하지 않습니다. 설정 방법은 다음과 같습니다.

```
node.master: false  
node.data: false  
#### or  
node.client: true
```

색인 성능 향상을 위한 전용 데이터 노드 구성

일반적으로 데이터 노드는 검색과 색인에 대한 모든 요청을 처리하므로 검색 로드 밸런서 노드를 분리하면 자연스럽게 데이터 노드의 성능 개선으로 이어지게 됩니다. 따라서 데이터 노드는 요청에 대한 코디네이션이나 응용 프로그램에 응답하는 등의 역할을 담당하지 않아도 되고, 노드에 속한 샤프의 색인과 검색 요청만 처리

하면 됩니다. 설정 방법은 다음과 같습니다.

```
node.master: false  
node.data: true
```

노드 관리 및 안정성 확보를 위한 마스터 노드 구성

마스터 노드는 기본으로 클러스터 관리를 담당하므로 상대적으로 검색 로드 밸런서 노드나 데이터 노드처럼 고사양의 장비가 아니어도 구성할 수 있습니다. 이는 장비에 대한 효율적인 배치를 가능하게 해줍니다. 설정 방법은 다음과 같습니다.

```
node.master: true  
node.data: false
```

안정성 확보를 위한 마스터 노드의 크기는 보통 Quorum 구성을 추천합니다.

[Quorum 구성 기본 계산식]

$$\text{Quorum} = (\text{마스터 자격 노드의 수}/2) + 1$$

다음은 `minimum_master` 설정을 위한 예로, `minimum_master`의 최소 크기는 2, 마스터 노드는 최소 3개 이상으로 구성해야 합니다. 이 예에서 마스터 노드를 2개로 구성하면 마스터 노드의 하나가 fail될 때 장애가 발생하므로 마스터 노드의 크기는 `minimum_master` 크기보다 항상 크게 설정해야 합니다.

[`minimum_master` 가 2일 때 노드 구성]

Total node size → 5대
`node.master: true` → 3대
`Quorum size = 3/2 + 1 → 2대`

Refresh 주기

인덱스에 대한 refresh로 NRT^{Near Real Time} 검색을 지원하는데, NRT 검색을 지

원하려면 refresh 주기를 1s로 설정해야 합니다. 색인 조건에 따라 이 값은 조정 해서 사용합니다. 하지만 너무 잦은 refresh 연산 사용은 많은 세그먼트 파일을 생성하게 되어 리소스가 낭비되고 세그먼트 병합^{Segments Merge} 작업으로 성능이 떨 어질 수 있으므로 주의해야 합니다.

index.refresh_interval: "1s"

스레드 풀

Elasticsearch에서 스레드 풀은 연산마다 역할이 나뉘어져 있어서 서비스 특 성에 맞춰 스레드 풀 크기를 조정해 주는 것이 리소스를 효율적으로 사용하는 방 법입니다. 성능을 테스트할 때 기본 설정값과 요건에 따른 수정값을 바탕으로 테 스트할 때 최적값을 찾을 수 있습니다. 가장 많이 사용하는 스레드 풀은 bulk, index, search 세 가지이며, 서비스 요건에 맞춰 설정을 변경합니다.

일반적인 스레드 풀 크기는 다음과 같이 정의합니다.

Thread pool size = cores x 2

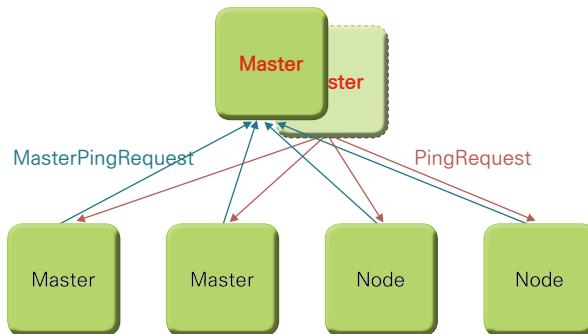
기본값은 Elasticsearch의 기본값⁰¹을 사용하고 최대값은 코어 수의 2배수까지 설정하는 것을 추천합니다. 너무 많은 스레드 풀 크기를 설정하면 대부분 자원을 낭비할 수 있으므로 유의하기 바랍니다.

Zen Discovery

Elasticsearch에서 노드 간의 통신을 관리하기 위해 사용하는 모듈입니다. 이 모듈이 성능에 영향을 미치는 요소는 다음과 같습니다.

01 Elasticsearch의 기본 스레드 풀 크기는 코어 수만큼 설정됩니다.

그림 7-2 노드 간 Zen Discovery



Ping

ping에는 모든 노드에서 마스터 노드로 보내는 master ping request와 마스터 노드에서 모든 노드로 보내는 ping request 두 가지가 있습니다. 이 두 가지 ping request를 가지고 마스터 노드 선출(Master Election)과 장애 감지(fault detection) 등의 작업을 수행합니다.

Multicast

모든 요청을 네트워크로 브로드캐스트^{broadcast}하는 방식입니다. 같은 네트워크에서 하나의 클러스터를 운영할 경우 클러스터를 쉽게 관리할 수 있지만, 자원 낭비를 초래할 수 있습니다.

Unicast

멀티캐스트와 다르게 설정된 노드 사이에서만 통신합니다. 멀티캐스트 방식보다는 유니캐스트 방식을 추천합니다.

Master Election

마스터 노드가 fail했을 때 다른 마스터 노드를 선출하는 것으로 Zen Discovery를 통해 이뤄지는 매우 중요한 작업입니다. 이는 장애를 예방하는 역할을 합니다.

마스터 노드 선출 과정은 다음과 같습니다.

- Step 1) 마스터 노드 목록을 정렬(기존 마스터 노드는 제외됨)
- Step 2) masters.get(0) 노드가 마스터로 선출
- Step 3) 각 노드로 선출된 마스터 정보 전송

Fault Detection

클러스터 안에 구성된 노드 중 장애가 발생한 노드를 감지하는 것으로, 장애가 발생한 노드의 샤프트에 재할당 Relocation 등의 작업을 수행합니다.

JVM 옵션

JVM 옵션 설정을 이용한 튜닝은 두 가지 방법이 있습니다. 하나는 Heap 크기 설정, 다른 하나는 GC 설정인데, Elasticsearch에서는 공식적으로 기본 GC 설정을 바꾸지 말도록 권고합니다.

Heap Size

Elasticsearch에서 사용 가능한 최대 메모리 크기는 32GB지만 Heap 크기가 너무 크면 Long GC를 유발할 수 있으므로 인덱스와 도큐먼트 크기, 캐시 크기 등을 고려하여 설정하는 것이 좋습니다.

[기본 Heap 크기 설정]

`-Xms256m -Xmx1g -Xmn1g`

다음은 Heap 크기에 따라 인덱스 버퍼의 크기가 얼마큼 설정되는지 보여줍니다. Heap 크기가 25GB일 때 인덱스 버퍼의 크기는 active 샤프트의 수(size)를 기준으로 계산되므로 샤프트당 $25\text{GB} \times 0.1 / 5 = 512\text{MB}$ 가 할당됩니다. 따라서 Heap 크기와 노드 내 active 샤프트의 수에 맞춰 설정하면 됩니다.

[인덱스 버퍼 크기]

`Index Buffer Size = (Heap Size × 0.1) / Active Shard Size`

GC

GC에서 기본으로 추천하는 설정은 CMSConcurrent-Mark and Sweep이며, G1GCGarbage First GC도 사용할 수 있습니다. 하지만 G1GC는 간혹 세그먼트 오류가 발생하기도 하고, 버그도 있으므로 사용할 때 주의해야 합니다.

JVM 옵션

```
-server -XX:+AggressiveOpts -XX:UseCompressedOops -XX:MaxDirectMemorySize -XX:+UseParNewGC  
-XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly
```

G1GC 설정 값

```
-XX:+UseG1GC
```

7.3.2 검색 튜닝

검색 튜닝, 즉 질의 최적화는 목적과 용도에 맞는 쿼리와 필터를 사용하는 것이 중요합니다. 하지만 그 이전에 도큐먼트에 대한 매팅 설정과 analyzer에 대한 최적화 작업이 선행되어 있다는 전제 하에 진행되어야 합니다. 검색 질의 최적화는 일반적인 방법이 아니고 서비스 종속적이므로 질의 특성을 잘 이해하고 구현해야 합니다. 여기서는 검색 질의 최적화에 활용 가능한 항목들과 이들이 튜닝에 어떤 영향을 주는지 살펴보겠습니다.

샤드

검색 질의 최적화에서 샤드는 두 가지로 활용할 수 있습니다.

기본 샤드 늘리기

기본 샤드를 늘리는 것은 멀티스레딩 처리로 성능 향상을 꾀하는 방법입니다. 하지만 개별 샤드에 저장된 도큐먼트 크기가 너무 작으면 샤드 1개로 구성된 인덱스의 질의 성능 보다 떨어질 수 있으므로 운영환경과 동일한 구성에서 최적값을 찾

는 것이 매우 중요합니다. 성능 테스트를 하지 않고 일반적인 기본 샤드의 최적값을 찾으려면 장비의 코어 크기와 같은 값으로 최적값을 구성하면 됩니다. 즉, 코어 크기가 24일 때 기본 샤드의 크기는 24가 됩니다.

[일반적인 기본 샤드 크기 설정]

```
index.number_of_shards: $CORE_SIZE
```

레플리카 샤드 늘리기

레플리카 샤드를 늘리면 검색 질의에 대한 처리량이 늘어나므로 성능이 향상됩니다. 하지만 저장소가 낭비될 수 있다는 점을 유의해야 합니다.

쿼리

쿼리의 사용은 기본적으로 도큐먼트에 대한 관련성(relevance) 요구가 있을 경우 사용하는데, 이때 다음 상황을 고려해야 합니다.

항상 같은 노드로 쿼리가 hitting되지 않게 한다

이는 클러스터를 구성한 노드 사이에 데이터 분산을 효율적으로 해서 리소스를 충분히 사용할 수 있게 해야 한다는 의미입니다. 즉, _id 값에 따른 도큐먼트의 샤드 할당이 잘 이루어지게 해야 합니다.

Zero Hit Query를 줄인다

쿼리는 필터와 다르게 캐시되지 않으므로 Zero Hit Query⁰²는 애플리케이션에서 처리하여 검색엔진까지 요청이 전달되지 않게 해야 한다는 의미입니다.

쿼리 결과를 캐시한다

인덱스 안의 도큐먼트가 빈번히 바뀌지 않는 경우 동일한 쿼리에 대한 결과를 애플리케이션에서 캐시 처리하면 성능 향상에 도움이 됩니다.

02 검색 질의 시 매칭 결과가 없는 쿼리를 의미한다.

Pagination에 주의한다

pagination은 다음 동작 방식 때문에 뒤로 갈수록 성능이 떨어질 수밖에 없습니다. 샤프트 5개, 1페이지에 50개의 도큐먼트를 보여줄 경우 ' $5 \times 1 \times 50 = 250$ 개'의 도큐먼트를 가져오고, 100페이지일 때는 ' $5 \times 100 \times 50 = 25000$ 개'의 도큐먼트를 가져오게 됩니다. deep pagination이 되면 성능이 저하되고 한 번에 리소스를 많이 사용하게 되어 안정성이 떨어지는 원인이 됩니다.

[Pagination 시 데이터 페치 계산식]

```
Data Fetch Size = number_of_shard x page x fetch_size  
#or  
Data Fetch Size = number_of_shard x ( from + size )
```

스크립트 사용 시 doc['field']를 사용한다

검색 결과에 대한 도큐먼트 필드값 접근 방법은 _source, _field, doc['field'] 가 있습니다. _source와 _field는 디스크 접근(disk access) 방식이고, doc ['field']는 캐시 접근(cache access) 방식이어서 doc['field']가 다른 두 방법 보다 접근 성능이 좋습니다.

필터

필터와 쿼리의 차이는 다음 표와 같습니다. 표에서 보듯이 관련성에 따른 도큐먼트 랭킹(scoring) 과정이 필요하지 않으면 필터를 사용하는 것이 성능상 좋습니다.

표 7-3 필터 vs. 쿼리

필터	쿼리
Binary yes/no	Relevance
Exact values	Fulltext
Cached	Not cached

7.3.3 색인 튜닝

색인 성능 튜닝은 장비, 네트워크, 문서 구조, 분석기 등과 같이 복합적인 요소에 영향을 많이 받습니다. 그러므로 실제 운영환경과 동일한 구성으로 색인 테스트를 수행해 보는 것이 매우 중요합니다. 다음은 색인 튜닝 시 확인해야 하는 기본 항목으로 작업 수행 시 검토해 보기 바랍니다.

샤드(레플리카, 크기)

색인 throughput을 늘리는 가장 쉬운 방법은 기본 샤드를 늘리는 것입니다. 또한, 색인 요청 시 레플리카 설정은 disable로 해야 원하는 색인 성능을 얻을 수 있습니다. 검색 튜닝에서도 언급했지만 색인에 최적화된 샤드 크기는 일반적으로 노드의 코어 수와 같습니다. 하지만 이 방법은 성능 테스트와 색인 특성에 따른 최적화 방법이 아니므로 유의해야 합니다.

또한, 샤드 크기 설정 시 데이터 노드의 크기도 중요한데, 기본 샤드는 데이터 노드의 크기보다 크거나 같아야 합니다. 이를 바탕으로 일반적인 색인 시 인덱스에 대한 샤드 크기의 최소값, 최대값은 다음과 같이 계산합니다.

최소값은 데이터 노드 크기보다 작아서는 안 됩니다.

[최소 샤드 크기]

index.number_of_shards: DATA_NODE_SIZE

최대값은 코어 크기보다 커서는 안 됩니다.

[최대 샤드 크기]

index.number_of_shards: CORE_SIZE

Bulk 요청 (refresh interval, request size, timeout 설정)

대용량의 데이터를 색인하는 데 Bulk API가 매우 유용합니다. Elasticsearch

에서 색인 요청을 최적화하는 방법은 멀티스레딩을 이용하여 작은 규모의 색인 요청을 여러 번 하는 것입니다. 다음은 bulk 요청에 대한 최적화 과정입니다.

- Step 1) **Disable replica & refresh_interval** : 복제와 리프레시 설정을 끕니다.
- Step 2) **Document read** : 색인할 문서를 읽습니다.
- Step 3) **Create bulk request** : 벌크 색인 요청을 생성합니다.
- Step 4) **Bulk request** : 벌크 색인 요청을 합니다.
- Step 5) **Repeat “Step 2”** : 이 과정을 “Step 2”부터 반복 합니다.
- Step 6) **Optimize** : 최적화 작업을 수행합니다.
- Step 7) **Enable replica & refresh_interval** : 복제와 리프레시 설정을 복구합니다.

Bulk 요청 크기

bulk 색인 요청을 할 때 요청 한 번에 대한 크기를 정의해야 합니다. 크기는 도큐먼트 크기 또는 물리적인 요청 크기를 기반으로 정의할 수 있습니다. 기본적으로는 1,000 ~ 5,000개 사이의 도큐먼트나 5 ~ 15MB 사이의 물리적인 크기로 정의합니다.

Bulk 요청 설정

bulk 색인 요청 시 처리 지연에 따른 오류는 몇 가지 설정으로 예방할 수 있습니다. bulk 스크립트 풀 크기를 조정하거나 클라이언트 애플리케이션에서 연결할 때 bulk 커넥션 풀 크기를 조정하거나 timeout 시간을 늘려주는 것입니다.

bulk 색인 요청은 서버 설정도 중요하지만 요청을 보내는 클라이언트 애플리케이션의 리소스 상황도 매우 중요합니다. 그러므로 bulk 색인 요청을 할 때에는 클라이언트/서버/네트워크 리소스 상황을 실시간으로 모니터링해야 합니다.

Exception 모니터링

bulk 색인 요청을 하다보면 종종 접하게 되는 예외에 EsRejectedExecutionException, TOO_MANY_REQUESTS, NoNodeAvailableException:

No node available, Timeout exception 등이 있습니다. 이는 요청이 과하거나 서버에서 처리할 수 있는 용량을 넘어선 것으로 판단하고 scale out을 고려하거나 요청을 줄여야 합니다.

Flush, Refresh 그리고 최적화

bulk 색인 요청 작업 완료 후 인덱스에 대한 최적화 작업을 진행하는데, 이 작업은 엄청난 리소스를 사용하므로 실제 운영환경에서 실시간으로 사용하면 안정성 부분에서 문제가 생길 수 있습니다. 하지만 최적화 작업은 색인 후 검색 성능 향상을 위해 필요한 작업이어서 처리를 안 할 수는 없습니다. 그래서 최적화의 대안으로 flush와 refresh를 활용합니다.

Optimize

bulk 색인으로 생성된 세그먼트 파일에 병합 작업을 수행합니다. 이 작업을 수행하면 검색 성능을 향상할 수 있습니다.

Flush

작업 로그(Translog)에 대한 커밋^{Commit}과 완료된 작업 로그 삭제(truncate)를 수행합니다.

Refresh

색인 데이터를 실시간으로 반영하기 위해 사용하며 refresh 호출 시 새로운 세그먼트 파일을 생성합니다.

7.4 정리

이번 장에서 다른 성능 최적화는 정답이 정해져 있지 않습니다. 서비스 요건과 장비 스펙, 데이터 특성에 따라 다양한 실험을 통해 최적 조건을 찾아야 합니다. 여기서 다른 내용을 기반으로 서비스 최적화 방법을 찾는데 도움이 되길 바랍니다.

지금까지 이 책을 읽어 주셔서 감사합니다. 잘못된 내용과 부족한 부분은 블로그나 커뮤니티 등을 통해 문의주시면 최대한 답변드리겠습니다.
끝까지 부족한 글을 읽어주셔서 다시 한 번 감사드립니다.