



실무에 활용하는 Elasticsearch 검색엔진 구축

3일차 : 아파치 루씬

오늘의 아젠다



루씬의 개요

루씬의 색인

루씬의 검색

루씬의 텍스트 분석

루씬의 고급검색

1. 루씬의 개요

루씬의 개요

정보검색 라이브러리(IR Library)

최초 더그 커팅 (Doug Cutting)이 개발 (2003년 3월 최초 릴리스/sourceforge)

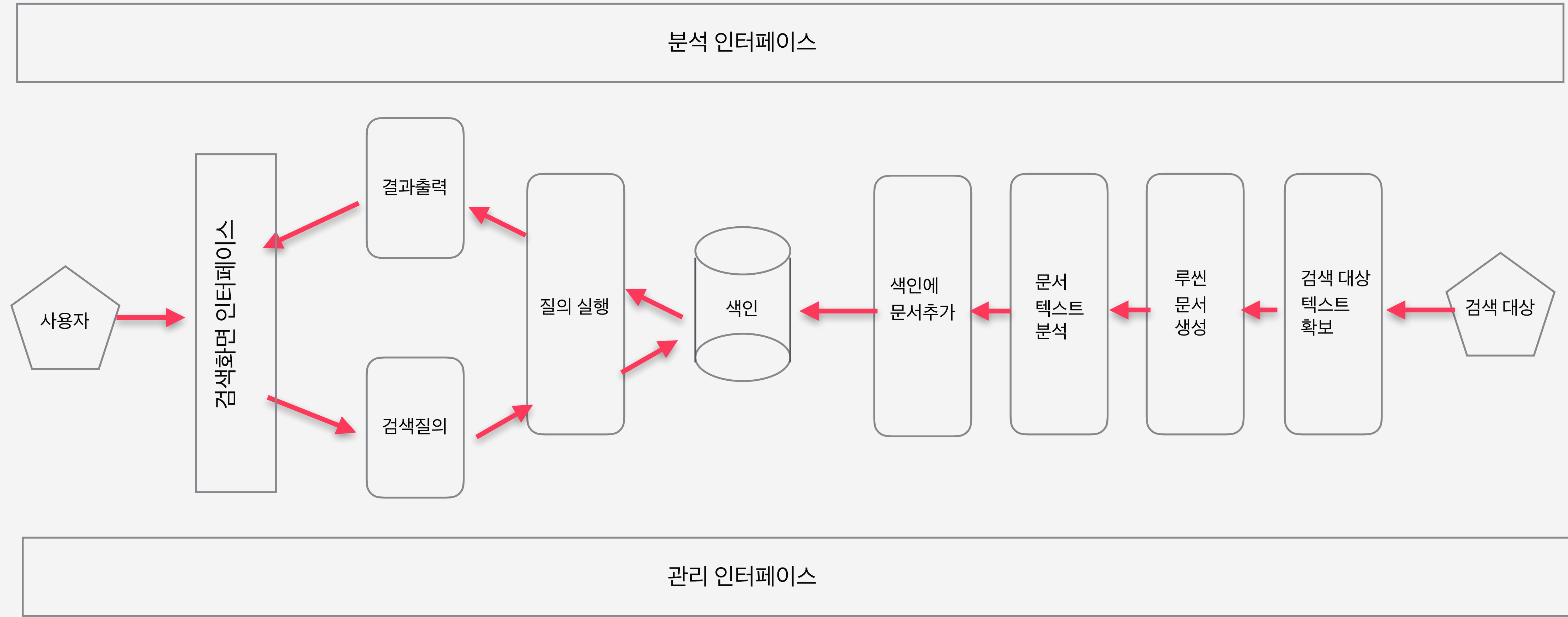
2001년 9월 아파치 재단에 합류

2005년 아파치 최상위 프로젝트 격상

lucene의 하위 프로젝트 너치(Nutch),솔라(solr)

Nutch의 하위 프로젝트 하둡(Hadoop)

루씬과 검색 어플리케이션의 구조



색인 관련 핵심 클래스

- IndexWriter

색인 과정에서의 가장 핵심 클래스

색인을 새로 생성하거나 기존 색인을 열고 문서를 추가,삭제,변경 기능 담당

- Directory

루씬의 색인을 저장하는 공간을 나타냄

클래스 자체는 추상 클래스 (상속 받아 메소드를 구현해야 함)

- Analyzer

텍스트를 분석하여 Term을 생성하는 역할

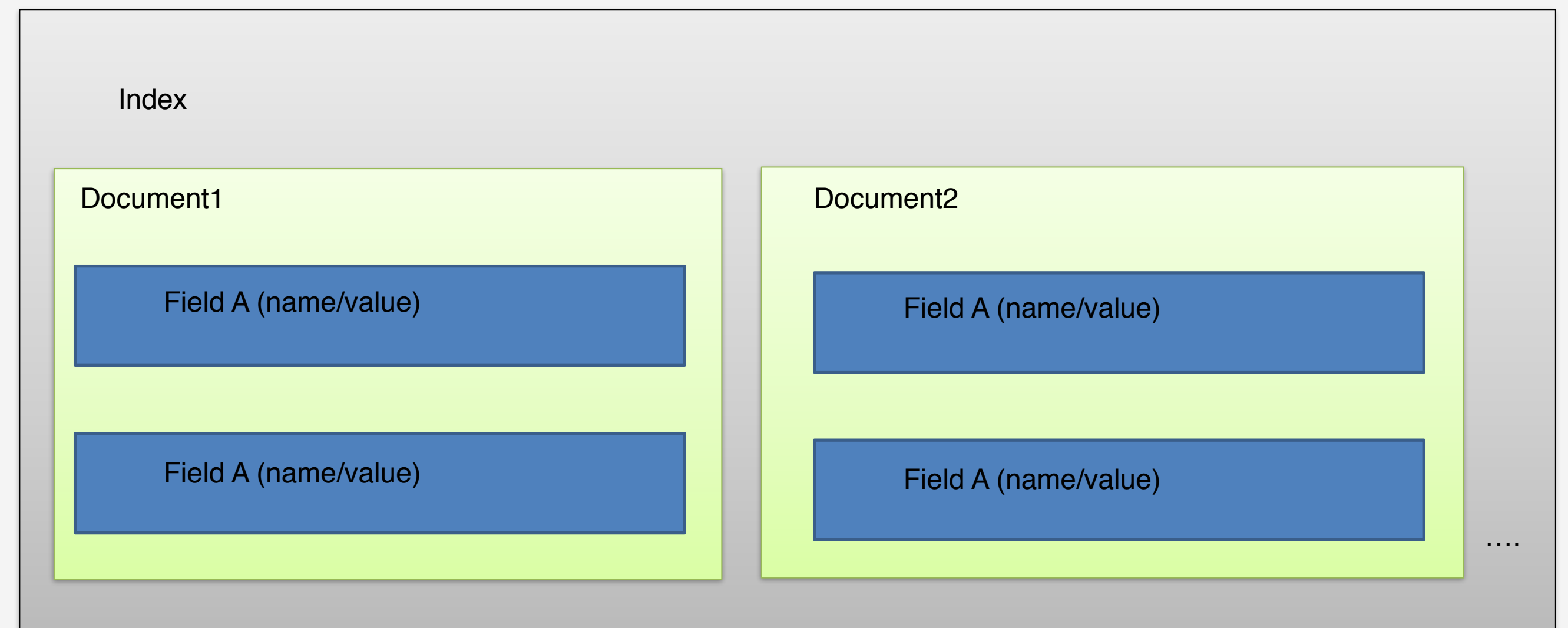
- Document

가상의 문서

검색 결과로 받아 보는 결과의 단위

- Field

Document는 Field로 구성되며 각자의 이름과 값이 있다.



검색 관련 핵심 클래스

- IndexSearcher

색인을 읽기 전용으로 열어서 사용 (Directory 인스턴스 지정)

색인을 새로 생성하거나 기존 색인을 열고 문서를 추가,삭제,변경 기능 담당

- Term

검색 과정을 구성하는 가장 기본 단위

색인 시점에서는 내부적으로 자동 생성 쿼리 시점에서는 Term을 생성하여 질의함

- Query

질의를 생성하는 클래스

하위 클래스로 TermQuery, BooleanQuery,PhraseQuery,PrefixQuery,TeramRangeQuery,FilteredQuery,SpanQuery 등으로 다양하게 제공됨

- TopDocs

검색 결과 중 최상위 N개의 문서에 대한 링크를 담고 있는 결과 클래스

색인관련 RDBMS와 비교 설명

Database	Lucene	설명
Table	Index	검색 색인
Column	Field	색인 필드
Row	Document	색인 문서 단위 데이터
Sql	Term	텍스트의 한 구절(word)를 나타냄
	Query	색인 질의 오브젝트
	Analyzer	색인 & 검색 언어 분석기
	QueryParser	색인 질의를 위한 파서 (Analyzer 부터 생성)
	Directory	색인 디렉토리 관리기
	Reader	색인 Reader (Directory 로 부터 생성)
	Searcher	색인 검색기 (Reader 로 부터 생성)
	IndexWriter	색인 생성기 (Directory & Analyzer 로 부터 생성)
insert	addDocument	IndexWriter.addDocuemnt(document) 도큐먼트 추가
update	updateDocument	IndexWriter.updateDocuemnt(term, document) 도큐먼트 수정
delete	deleteDocument	IndexWriter.deleteDocuemnt(delTerm, document) 도큐먼트 삭제
PrimaryKey	uniqueId	도큐먼트의 고유 ID (추상적인 필드)
commit	commit	데이터 업데이트 commit

루씬의 데이터 모델의 특징

필드(Field) 고려 사항

- 필드의 내용을 색인 할것인가? 하지 않을 것인가?
- 색인을 할 경우 텀벡터(Term Vector)를 색인할것인가? 하지 않을것인가?
- 색인의 내용을 색인 할것인가? 하지 않을것인가?

기본적으로 루씬은 Schemaless 모델

색인에 추가하려고 준비하는 문서의 객체가 완전히 다른 필드 구조여도 상관 없음

필드의 개수와 이름이 맞지 않아도 됨

기본적으로 루씬의 문서는 평평한 1차원 데이터 모델

RDBMS와 같이 JOIN 연산을 통해 관계를 지정하거나 내용을 중첩시킬수 없음 (But Solr와 E/S 는 가능함)

루씬의 다운로드 및 설치

루씬은 하나의 .jar 파일이며 기본 자바 프로젝트에 /lib 에만 존재해도 언제든지 사용 가능

<http://apache.tt.co.kr/lucene/java/6.3.0> 다운로드

← → ↻

Index of /lucene/java/6.5.1

Name	Last modified	Size	Description
Parent Directory		-	Java-Apache (old)
changes/	27-Apr-2017 02:50	-	Java-Apache (old)
lucene-6.5.1-src.tgz	21-Apr-2017 20:50	31M	Java-Apache (old)
lucene-6.5.1.tgz	21-Apr-2017 20:50	65M	Java-Apache (old)
lucene-6.5.1.zip	21-Apr-2017 20:50	76M	Java-Apache (old)

Apache Server at apache.tt.co.kr Port 80

루씬 소스코드

바이너리 파일

▶ analysis	그저께 오후 10:29	--	폴더
▶ backward-codecs	그저께 오후 10:29	--	폴더
▶ benchmark	그저께 오후 10:29	--	폴더
CHANGES.txt	2017년 4월 21일 오전 6:52	643KB	일반 텍스트
▶ classification	그저께 오후 10:29	--	폴더
▶ codecs	그저께 오후 10:29	--	폴더
▶ core	그저께 오후 10:29	--	폴더
▶ demo	그저께 오후 10:29	--	폴더
▶ docs	2017년 4월 21일 오후 7:19	--	폴더
▶ expressions	그저께 오후 10:29	--	폴더
▶ facet	그저께 오후 10:29	--	폴더
▶ grouping	그저께 오후 10:29	--	폴더
▶ highlighter	그저께 오후 10:29	--	폴더
▶ join	그저께 오후 10:29	--	폴더
JRE_VERSION_MIGRATION.txt	2017년 4월 21일 오전 6:51	2KB	일반 텍스트
LICENSE.txt	2017년 4월 21일 오전 6:51	25KB	일반 텍스트
▶ licenses	2017년 4월 21일 오전 6:52	--	폴더
▶ memory	그저께 오후 10:29	--	폴더
MIGRATE.txt	2017년 4월 21일 오전 6:52	4KB	일반 텍스트
▶ misc	그저께 오후 10:29	--	폴더
NOTICE.txt	2017년 4월 21일 오전 6:51	9KB	일반 텍스트
▶ queries	그저께 오후 10:29	--	폴더
▶ queryparser	그저께 오후 10:29	--	폴더
README.txt	2017년 4월 21일 오전 6:51	724바이트	일반 텍스트
▶ replicator	그저께 오후 10:29	--	폴더
▶ sandbox	그저께 오후 10:29	--	폴더
▶ spatial	그저께 오후 10:29	--	폴더
▶ spatial-extras	그저께 오후 10:29	--	폴더
▶ spatial3d	그저께 오후 10:29	--	폴더
▶ suggest	그저께 오후 10:29	--	폴더
SYSTEM_REQUIREMENTS.txt	2017년 4월 21일 오전 6:51	731바이트	일반 텍스트
▶ test-framework	그저께 오후 10:29	--	폴더

루씬 코어

이외의 다양한 contrib 프로젝트로
추가 기능들이 개발되고 있음

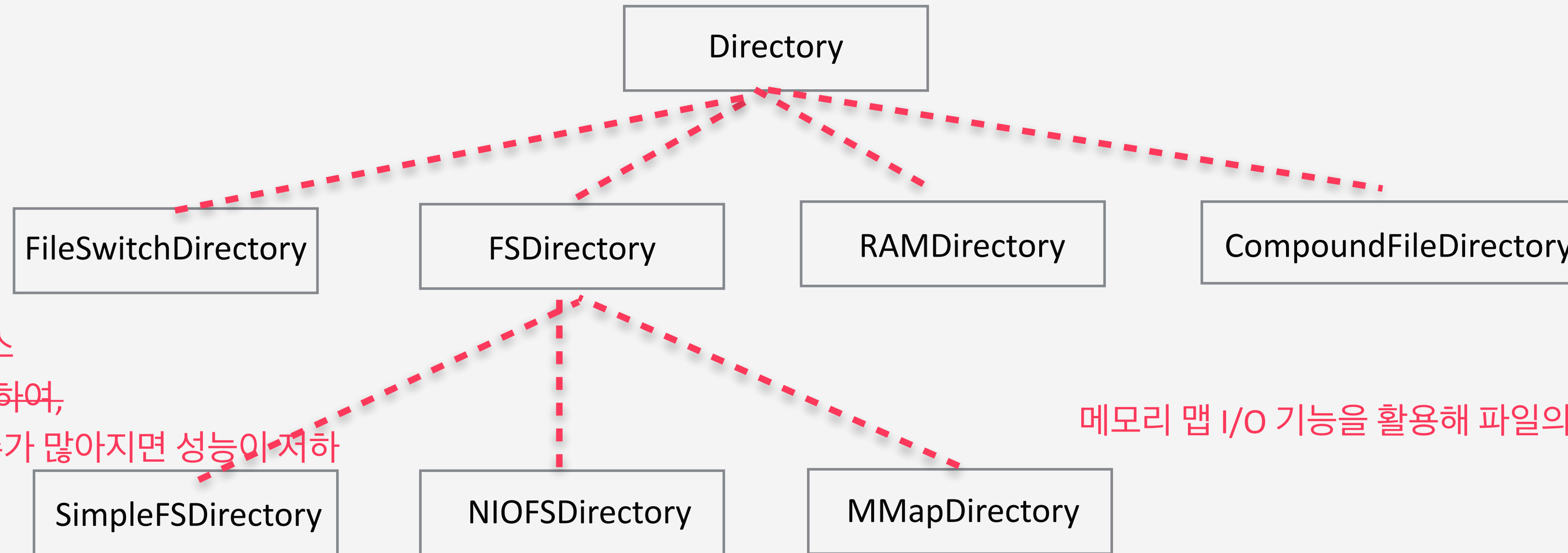
다른 언어에서의 루씬

포팅종류	설명	포팅 프로젝트	장점	단점
네이티브 포팅	루씬의 소스코드를 모두 대상 프로그래밍 언어로 재작성	<u>lucene.net</u> (C#) Cluene(C++) KinoSearch(C,Perl) Ferret (Ruby) Lucy (Perl)	전반적으로 가볍다 대상언어에서 제공하는 기능을 직접적으로 사용	포팅 작업이 어렵다. (대부분 프로젝트가 버전업이 되지 않는다.) 버그 발생확률이 높다
역네이티브 포팅	대상 언어를 JVM에서 실행 작성한 언어를 JVM에서 구동	Jython JRuby	전반적으로 가볍고 루씬과 100% 호환	대상언어가 JVM 에서 구동되기 때문에 대상언어의 확장 기능을 사용 못할 가능성 있음
로컬 래퍼	JVM 자체를 대상 언어의 실행환경에 내장하고 JVM과 루씬에 대한 인터페이스를 래핑하여 루씬 이 기능을 사용	PyLucene	루씬의 API만 노출하기 때문에 포팅 작업이 빠르다 루씬과 100% 호환	두개 언어 환경을 실행하기 때문에 무겁다
클라이언트 서버	자바 버전의 루씬을 실행하여 표준 프로토콜을 통해서 루씬의 기능을 제공(각 프로세서를 따로 실행)	solrj php client E/S Rest Client	클라이언트 라이브러리를 손쉽게 사용 루씬 기능 이상의 기능을 사용 가능	관리 서버가 하나 더 늘어난다.

루씬의 색인

Directory 종류

추상클래스



가장 기본적인 Directory 클래스
java.io.*패키지의 기능을 활용하여,
동시에 사용하려는 스레드 개수가 많아지면 성능이 저하

메모리 맵 I/O 기능을 활용해 파일의 내용일 읽어온다.

색인을 파일 시스템에 저장하지만 java.nio.* 패키지의 기능을 활용
동시에 사용하려는 스레드의 개수가 많아져도 성능이 크게 떨어지지 않음

IndexWriter의 주요 메소드

writer.addDocument(문서)

writer.updateDocument(문서)

writer.deleteDocuments(쿼리 혹은 텀)

writer.commit()

색인을 반영함

writer.forceMerge()

색인을 강제 병합함

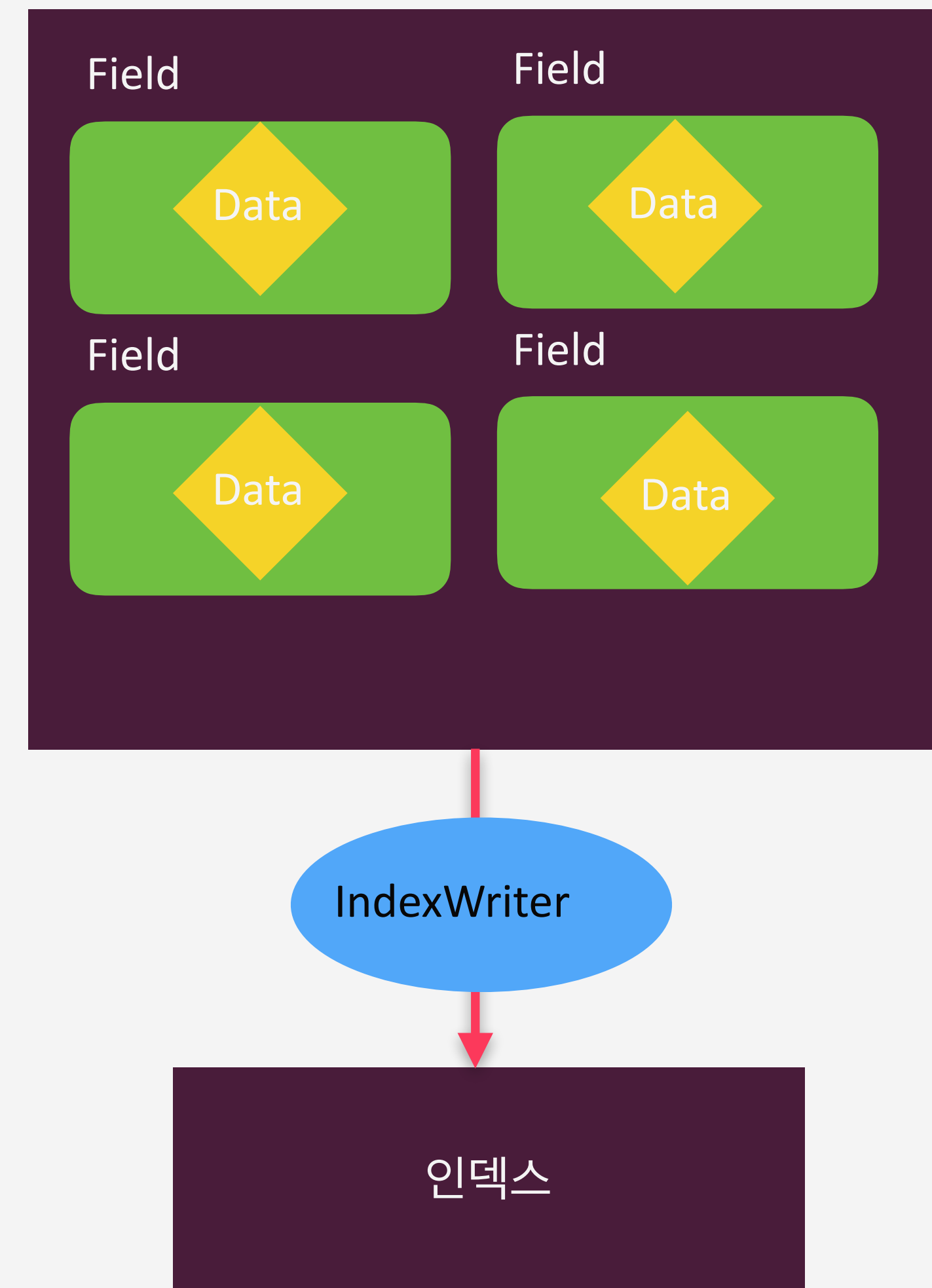
writer.rollback()

색인을 반영하지 않고 되돌림

writer.close()

writer를 다음

문서(Document)



Field 관련 주요 속성

StringField : 저장용 문자열 필드 (토크나이징 하지 않는다)

TextField : 풀텍스트 검색용 문자열 필드 (토크나이징을 실시 한다)

NumericField : IntField, LongField, FloatField, DoubleField (Deprecated)

IntPoint, LongPoint, FloatPoint, DoublePoint (new)

DocValuesField : 도큐먼트에 한개씩 Scoring & Sorting을 위한 필드

ByteDocValueField, DerefBytesValuesField, DoubleDocValuesField, FloatDocValuesField,
IntDocValuesField, LongDocValuesField, PackedLongDocValuesField, ShortDocValuesField,
SortedBytesDocValuesField, StraightBytesDocValuesField

DateField : 날짜 저장 필드 (Deprecated)

Field 설정 방법

```
FieldType ft = new FieldType(TextField.TYPE_STORED);  
ft.setIndexOptions(IndexOptions.DOCS_AND_FREQS_AND_POSITIONS);  
//색인 시 빈도 및 포지션 저장여부  
ft.setOmitNorms(true); //길이 정규화와 Index Time 부스팅 사용 여부 (메모리 절약 효과)  
ft.setStored(true); //데이터 저장 여부  
ft.setStoreTermVectorOffsets(true); //터미벡터 오프셋 저장 여부  
Document doc = new Document();  
doc.add(new Field("title", "나는 자랑스러운", ft));
```

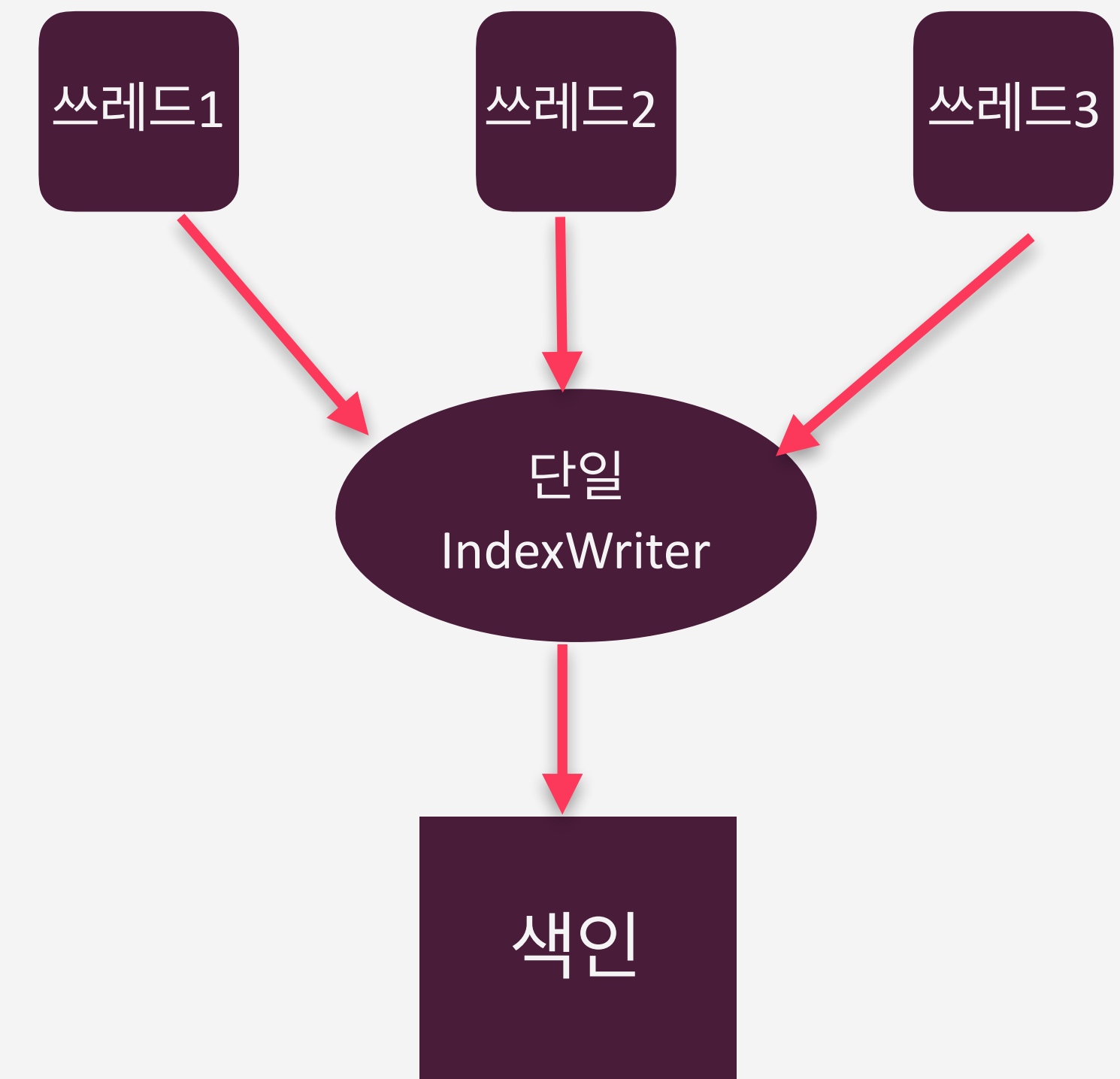

루씬의 ACID 트랜잭션과 색인의 일관성

루씬은 기본적으로 ACID 트랜잭션 모델
오직 IndexWriter를 통해 하나의 트랜잭션만 진행

Atomic(원자성)	IndexWriter가 변경한 내용은 모두 색인에 반영하거나 모두 반영되지 않음
Consistency(일관성)	색인은 항상 일관성을 유지 updateDocument 메소드는 반드시 삭제 이후 AddDocument 메소드 실행
Isolation(고립성)	IndexWriter가 추가나 삭제를 하여도 커밋전에는 IndexReader가 볼수 없음
Durability(지속성)	어플리케이션이 예외를 처리 못하거나 JVM 및 서버가 비정상적으로 종료되어도 색인은 항상 정상 상태를 유지(최종 성공 커밋 상태 유지)

루씬의 병렬처리 , 쓰레드 안정성, Lock

- 1) 특정 색인에 대해 읽기 전용의 IndexReader는 몇개라도 열어 사용할 수 있음 (다른 장비에서도)
성능과 시스템 자원 활용의 측면에서 하나의 IndexReader 인스턴스를 생성후 여러 쓰레드에서 IndexReader를 공유해 사용 권장
여러 쓰레드에서 하나의 IndexReader 인스턴스를 통해 검색 가능
- 2) 색인 하나에 대해 IndexWriter는 하나만 열수 있음
루씬에서 쓰기 락을 사용해 IndexWriter를 하마나 열게 제한(파일 기반의 락 사용 - write.lock)
indexWriter인스턴스가 생성되자마자 쓰기 락을 확보
해당 IndexWriter를 닫음으로서 쓰기 락이 해제
- 3) indexReader는 indexWriter가 색인의 내용을 변경하고 있는 도중이라도 언제든지 열어 사용할수 있음
IndexReader는 변경 작업과 관계없이 열리는 시점의 색인을 표현
IndexWriter가 진행한 변경 사항을 반영하고 IndexReader를 새로 열기 전에는 변경 사항이 보이지 않음
- 4) IndexReader나 IndexWriter인스턴스는 여러 쓰레드에서 얼마든지 공유해 사용 가능



루씬의 병렬처리 , 스레드 안정성, Lock (계속)

1. NativeFSLockFactory

FSDirectory 클래스에서 사용하는 기본 락

java.nio패키지를 통해 운영체제에서 제공하는 락 기능을 활용

2. SimpleFSLockFactory

자바의 File.createNewFile 메소드를 사용해 락 파일을 생성

NativeFsLockFactory보다 시스템의 종류에 영향을 덜 받음

하지만 JVM이 비정상적으로 종료되거나 JVM이 종료 할때 IndexWriter를 제대로 닫지 못한다면 write.lock파일이 남아있을수 있음 (수동 제거)

4. SingleInstanceLockFactory

락을 메모리 안에 생성

RAMDirectory를 사용하는 경우 기본 설정

모든 IndexWriter가 동일한 JVM안에서 생성될때만 제대로 사용

5. NoLockFactory

락을 사용하지 않는다. 루씬이 제공하는 락을 사용할 필요가 없다고 판단되는 경우에만 사용

IndexWriter에서 락을 확보 했다면 그 이후에 생성하려는 IndexWriter는 단순히 기본 설정으로 1초마다 락을 확보하려고 재시도

기존에 락을 갖고 있던 IndexWriter가 락을 해제하면

즉시 신청한 순서대로 다음 IndexWriter에 락을 넘겨주 등의 큐는 없음

색인의 병합

세그먼트의 수가 많아지면 IndexWriter에서 세그먼트를 병합

병합의 장점

- 세그먼트의 개수가 줄어 들어 운영체제의 파일 개방 개수 제한에 유연히 대처
- 디스크 공간 소요량 감소
- 검색 속도 향상

병합 방법1 : 강제 병합

`writer.forceMerge();`

병합 방법2 : 스케줄링

MergeScheduler를 이용하여 주기적 병합

\

색인의 병합

MergeFactor에 의한 세그먼트의 병합 전략

세그먼트를 병합하는 빈도와 크기를 제어

<http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html>

LogByteSizeMergePolicy(mergeFactor,minmergeSize)

색인의 바이트 용량을 기준으로 색인을 병합하는 전략

LogDocMergePolicy(mergeFactor,minmergeCount)

색인의 문서 갯수를 기준으로 색인을 병합하는 전략

TieredMergePolicy (maxMergeAtOnce,maxMergedSegmentBytes)

상기 LogMerge 전략은 세그먼트수가 지수적으로 증가 하기 때문에 병합 처리 비용을 무시 할 수 없음
병합처리전에 인덱스의 삭제 문서도 불필요한 공간을 차지

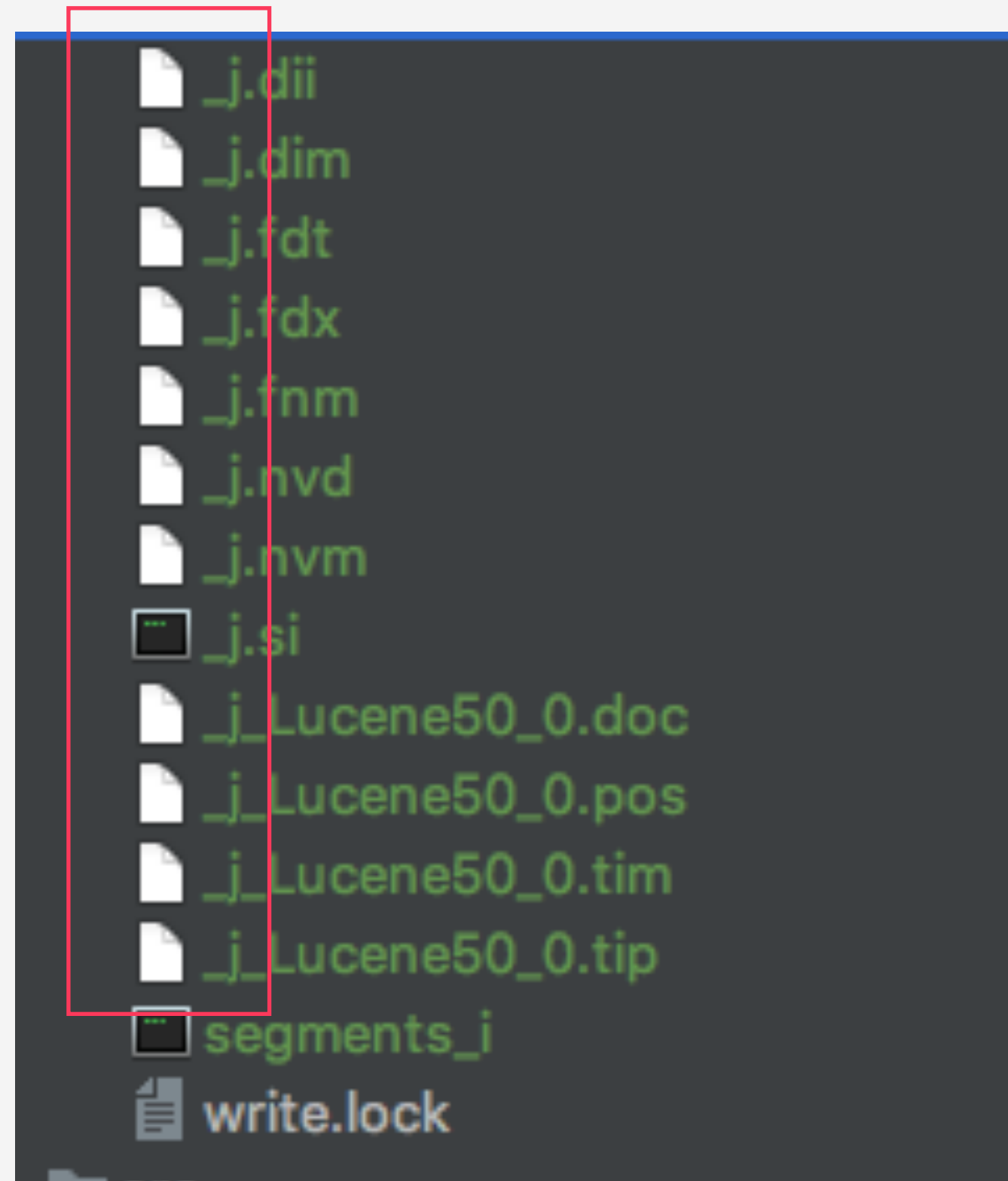
계층적으로(내림차순)이 되도록 병합대상 세그먼트를 (추출 방법은 독자 점수 모델)

maxMergeAtOnce : 1 회 병합 처리에서 최대 세그먼트수

maxMergedSegmentBytes : 병합후 최대 크기

색인의 파일 구조

색인 세그먼트(segment)



prefix 가 같으면 하나의 세그먼트



최적화 되지 않은 색인의 모습

색인의 파일 구조

필드 이름 (.fnm)

- 필드의 내용 색인여부
- 텀벡터 사용여부
- Norm 값 저장 여부

텀 사전(tim, tip)

필드와 값의 쌍값을 저장함 (글자순 정렬)

텀마다 빈도수를 저장

.tim(Term Dictionary) : 용어사전으로 용어 정보를 저장

.tip (Term Index) : 용어 사전에서의 인덱스

(색인내에서도 빠르게 텀을 찾아가기 위해서 내부에 역색인을 한번더 만듦)

텀 빈도수(.doc)

각 문서안의 텀의 빈도수 저장

텀 위치 (.pos)

문서내에서 각 텀의 위치 보관

구문 질의나 Span 질의시에 활용됨

.frm

필드이름	색인여부	Norm 여부
subject		
contents		
modified		
title		
category		
isbn		
path		
author		
url		

.tim

필드	값	문서빈도
author	Andy Hunt	1
	Bob Flaws	1
category	/education/...	1
	/health/..	1
contents	action	3
	junit	2
isbn	53344	1
modified	tajimara	2
path	/User/Jihoon/Doc	1
pubmonth	197708	1
subject	agile	2
title	action	3

.doc

문서번호	빈도수
...	
5	1
6	2
...	

.pos

위치
...
9
1
...

색인의 파일 구조

원문 저장(.fdx .fdt)

.fdx에 간단한 색인 정보 입력
.fdx를 기준으로 fdt 값을 찾아냄

Per-Document Values (.dvd,dvm)

추가적인 스코어 요소나 개별 문성의 정보 표현

페이로드 저장(.pay)

Payload값과 Offset 값 저장

텀 벡터(.tvf .tvd . tvx)

.tvf : 알파벳 순서대로 텀과 빈도수, 시작 지점과 위치 정보 저장
.tvd : 특정문서에서 텀벡터를 갖고 있는 필드 목록 저장, .tvf 파일에 저장된 바이트 단위의 위치를 함께 저장
.tvx : 문서 번호에서 .tvf와 .tvd 파일을 직접 이동할 수 있는 색인 정보 저장

Norm(.nvd, .nvm)

색인 과정에서 확보한 가중치 값을 저장
문서 가중치,필드가중치,본문길이 정규화 정보

삭제된 문서 (.del)

삭제된 문서에 해당하는 비트를 1로 설정 저장

Segment Info(.si)

세그먼트에 대한 메타정보를 저장

루씬의 검색

핵심 검색 API

클래스	용도
IndexSearcher	색인 검색하는 기능을 담당하는 클래스 모든 검색은 IndexSearcher 인스턴스의 여러가지 search 메소드로 진행
Query(하위 클래스 포함)	실제 Query 하위 클래스에서 특정 질의를 구현 IndexSearch 의 search 메소드는 Query 객체를 인자로 받음
QueryParser	사람이 직접 입력하고 사람이 알아 볼수 있는 텍스트 형태의 질의를 Query 객체로 변환
TopDocs	연관도가 높은 결과 문서를 담는 클래스
ScoreDoc	TopDocs 클래스에 담긴 하나의 클래스

다양한 질의 종류

TermQuery

```
Term t = new Term("title","대통령");  
Query query = new TermQuery(t);  
TopDocs docs = search.search(query,10);
```

TermRangeQuery

NumericRangeQuery

PrefixQuery

```
PrefixQuery query = new PrefixQuery(t);
```

BooleanQuery

PhraseQuery (사용 빈도 높음)

텀의 일정거리 안에 존재하는 검색 일치만

```
PharaseQuery query = new PharaseQuery();  
query.setSlop(slop); //텀 사이의 허용 거리
```

```
for(String word : phase) {  
    query.add(new Term("title",word));  
}
```

WildcardQuery

```
Query query = new WildecardQuery(new Term("title","?ild*"));
```

FuzzyQuery

MatchAllDocsQuery

QueryParser의 질의 표현식

+ : 반드시 존재해야 하는 텀
예) title : +대통령 +문재인 = title : 대통령 AND 문재인

[공백] : 반드시 존재하지 않아도 되는 텀

- : 존재하지 말아야 하는 텀

[A TO B] : A부터 B값
예) pubdate:[20160101 TO 20171010]

“(따옴표) : PhaseQuery

^ : 질의의 중요도 결정

junit^2 testing => junit 이라는 텀의 질의의 중요도가 2

title:junit^2 content: junit => title 필드의 질의 중요도가 2

문자열 형식	기호 형식
a AND b	+a +b
a OR b	a b
a AND NOT b	+a -b

QueryParser는 다양한 특수 문자를 사용해 질의 종류와 속성을 지정

\ + - ! () : ^] { } ~ * ? 는 반드시 escape 처리 해주어야 함

루씬에서의 스코어 계산

Lucene은 정보 검색 모델중 Boolean 모델과 벡터스페이스 모델이 결합된 형태로 결과를 도출
Boolean 모델로 결과를 한정하고 벡터스페이스 모델로 스코어를 부여

벡터스페이스 모델에서의 문서와 쿼리는 다차원 공간에서 가중치 벡터로 표현됨
각 텀은 차원이고 가중치는 TF-IDF 값

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|} = \frac{\sum_{i=1}^n A_i \times B_i}{\sqrt{\sum_{i=1}^n (A_i)^2} \times \sqrt{\sum_{i=1}^n (B_i)^2}}$$

가중치 벡터의 내적(스칼라의 곱) 길이가 정규화된 가중 벡터의 내적으로 간주
두 벡터의 정규화된 길이

루씬이 추가한 컨셉

- 1) 단위벡터로 정규화 한것에 문제가 있다. (doc-len-norm(d))
- 2) 색인을 생성할때 특정 문서가 더 중요할 수 있다. (doc-boost(d))
- 3) 루씬은 필드기반이므로 필드별 부스트도 따로 있어야 한다.
- 4) 쿼리에서도 용어별로 가중치가 다르다 (query-boost(q))
- 5) 문서는 다차원 쿼리와 일치 할수 있다(coord-factor(q,d))

$$\text{score}(q, d) = \text{coord-factor}(q, d) \cdot \text{query-boost}(q) \cdot \frac{V(q) \cdot V(d)}{|V(q)|} \cdot \text{doc-len-norm}(d) \cdot \text{doc-boost}(d)$$

Lucene 개념적 채점 공식

루씬에서의 스코어 계산

TF(Term Frequency) : $\sqrt{\text{freq}}$
문서에서 해당 Term이 나온 횟수

IDF(Inverse Document Frequency) : $\log(\text{numDocs}/(\text{docFreq}+1)) + 1$
전체 문서에서 해당 Term이 나온 횟수의 역수

Coord : $\text{overlap} / \text{maxOverlap}$
검색된 문서에서 쿼리의 Term이 몇 개 들어있는지에 대한 값
OR 검색에서만 효과
예) 나이키 운동화 검색 => 나이키 매장 = coord 값은 1/2

lengthNorm : $1/\sqrt{\text{numTerms}}$
문서길이 대비 중요도

queryNorm : $1/\sqrt{\text{sumOfSquaredWeights}}$
문서간의 비교에서 직접적 영향 없는 값 (하나의 쿼리에서는 동일한 값)
쿼리간의 비교를 위한 정규화 값

$$\text{score}(q,d) = \text{coord}(q,d) \cdot \text{queryNorm}(q) \cdot \sum_{t \text{ in } q} (\text{tf}(t \text{ in } d) \cdot \text{idf}(t)^2 \cdot t.\text{getBoost}() \cdot \text{norm}(t,d))$$

Lucene Practical Scoring Function

$\text{score}(q,d) =$
 $\text{Sigma}(t \text{ in } q) (\text{tf}(t \text{ in } d) * \text{idf}(t)^2 * \text{getBoost}(t \text{ in } q) * \text{getBoost}(t.\text{field} \text{ in } d) * \text{lengthNorm}(t.\text{field} \text{ in } d))$
 $* \text{coord}(q, d)$
 $* \text{queryNorm}(q)$

$\text{lengthNorm}(t.\text{field} \text{ in } d) = 1/\sqrt{\text{numTerms}} * f.\text{getBoost} * d.\text{getBoost}$
 $\text{queryNorm}(q) = \text{queryNorm}(\text{sumOfSquaredWeights})$

sumOfSquaredWeights = $\text{Sigma}(t \text{ in } q)(\text{idf}(t) * \text{getBoost}(t \text{ in } q))^2$

검색 스코어 계산 예제

문서1: title필드 : 현대/카드/각/좋다 (텀갯수 3개)
content필드 : 현대/카드/는/매우/좋은/카드/입니다.(전체 텀갯수 5개)
문서가중치 : 3
문서2: title필드 : 현대/자동차 (텀갯수 2개)
content필드 : 현대/자동차/는/자동차/를/만듭니다.(전체 텀갯수 4개)
문서가중치 : 5

필드별 가중치 : title(10) , content(1)

질의어 : 현대^3 OR 카드

문서1의 스코어 계산

텀의 빈도

전체문서수

텀의문서출현빈도

텀가중치

텀의갯수

문서가중치

필드가중치

title에서 “현대”

$\text{sqrt}(1) \times (\log(2/(2+1)) + 1)^2 \times 3 \times (1/\text{sqrt}(3) \times 3 \times 10)$

+

title에서 “카드”

$\text{sqrt}(1) \times (\log(2/(1+1)) + 1)^2 \times (1/\text{sqrt}(3) \times 3 \times 10)$

+

content에서 “현대”

$\text{sqrt}(1) \times (\log(2/(2+1)) + 1)^2 \times 3 \times (1/\text{sqrt}(5) \times 3 \times 1)$

+

content에서 “카드”

$\text{sqrt}(2) \times (\log(2/(1+1)) + 1)^2 \times (1/\text{sqrt}(5) \times 3 \times 1)$

35.2728135569

텀가중치가 “현대”가 높기 때문에 “카드”보다 더 높은 점수 받음

17.3205080757

2.73222038959

필드가중치가 낮기때문에 title보다 낮은 점수를 받음

1.8973665961

coord

$2/2 \times 1 \times 35.2728135569 + 17.3205080757 + 2.73222038959 + 1.8973665961 = 57.2229086183$

검색 스코어 계산 예제

문서1: title필드 : 현대/카드/각/좋다 (텀갯수 3개)
content필드 : 현대/카드/는/매우/좋은/카드/입니다.(전체 텀갯수 5개)
문서가중치 : 3
문서2: title필드 : 현대/자동차 (텀갯수 2개)
content필드 : 현대/자동차/는/자동차/를/만듭니다.(전체 텀갯수 4개)
문서가중치 : 5

필드별 가중치 : title(10) , content(1)

질의어 : 현대^3 OR 카드

문서2의 스코어 계산

텀의 빈

전체문서

텀의문서출현빈

텀가중

텀의갯

문서가중

필드가중

Σ

title에서 “현대”
sqrt(1) X (log(2/(2+1)) +1)² X 3 X (1/sqrt(2) * 5* 10)

+

title에서 “카드”
카드에서 히트된 텀이 없음

+

content에서 “현대”
sqrt(1) X (log(2/(2+1)) +1)² X 3 X (1/sqrt(4) * 5* 1)

+

content에서 “카드”
카드에서 히트된 텀이 없음

72.000329172
문서의 가중치가 문서1보다 높기 때문에 스코어 높


0

5.0911921005

0

coord

1/2 * 1 * 117.57604519+57.735026919+11.757604519 +6.32455532034 = 38.5457606364

Fast campus

데이터 사이언스

문서2가 문서 가중치에 의해서 높은 점수를 받았으나 TF-IDF 계산과 coord 계산으로 인해 문서1이 좀더 적합만 문서라고 스코어링됨

검색 스코어 계산 요약정리

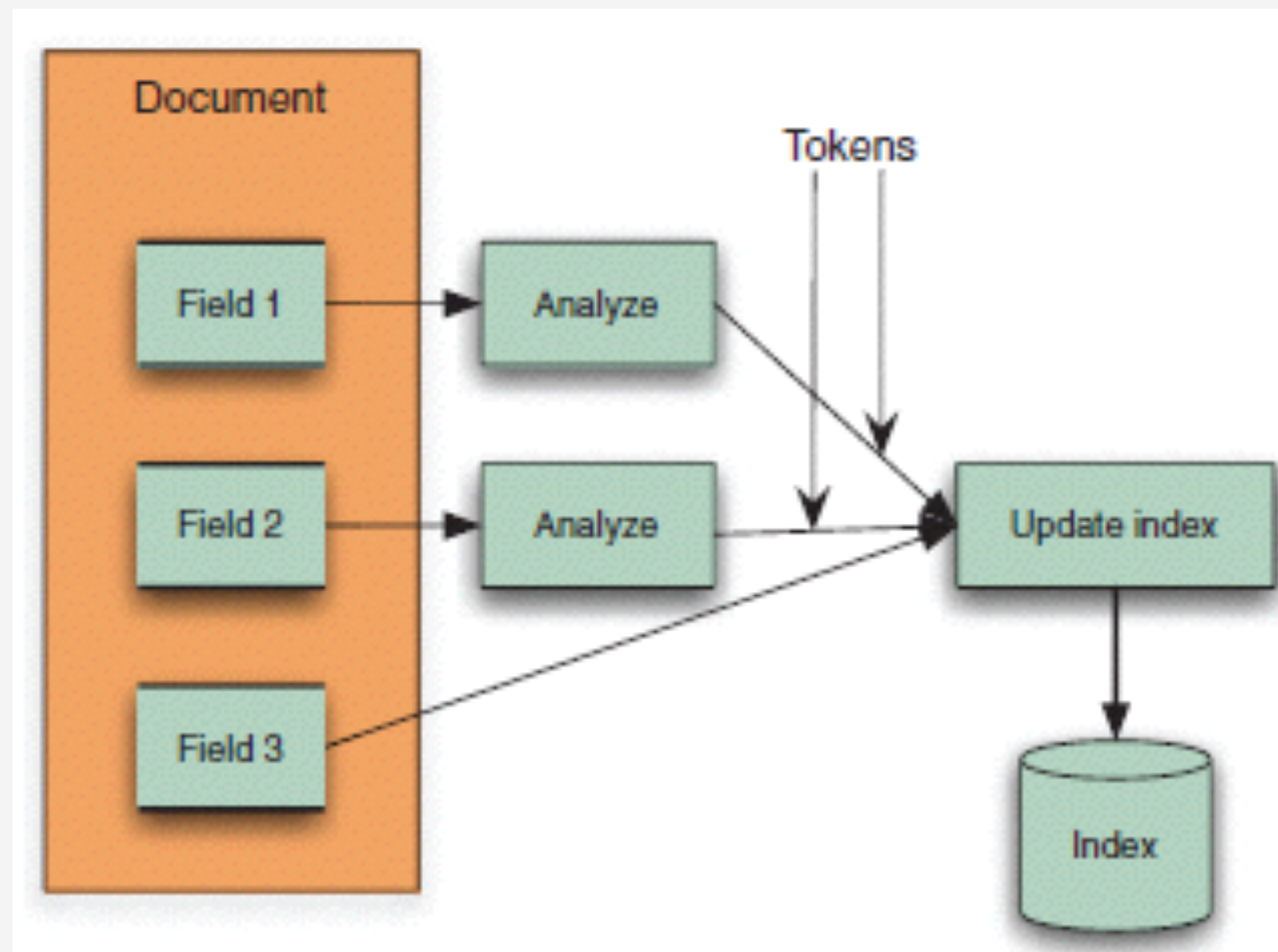
- 1) 검색어(term)가 많이 일치할수록 랭킹이 높아진다. (Term Frequency 증가)
- 2) 검색어중에 흔한 단어의 경우 가중치가 낮아진다. (두 단어 이상으로 검색될시 의미 있음)
- 3) 여러 필드를 검색시에 가중치를 높게준 필드에 일치 할수록 랭킹이 높아진다.(필드별 가중치 적용)
- 4) 데이터가 짧은 필드에서 일치된 것과 데이터가 긴 필드에서 일치된 것중 짧은 쪽이 랭킹이 높아진다. (length Norm 적용)
- 5) 두개 이상의 단어 검색시에 Term 가중치를 줄 경우 Term 가중치를 받은 Term이 일치할 경우 랭킹이 높아진다.
(Term Weight 적용)
- 6) 검색어가 반복적으로 입력될경우 Term 가중치와 비슷한 효과가 있다.

예) love love love hate 라는 검색어는 love라는 단어에 3배의 가중효과를 부여한다.
- 7) 색인 시점에서 문서에 대한 가중치를 부여할경우 부여된 문서의 랭킹이 높아진다(문서 가중치 적용)

루씬의 텍스트 분석

루씬에서의 형태소 분석의 개요

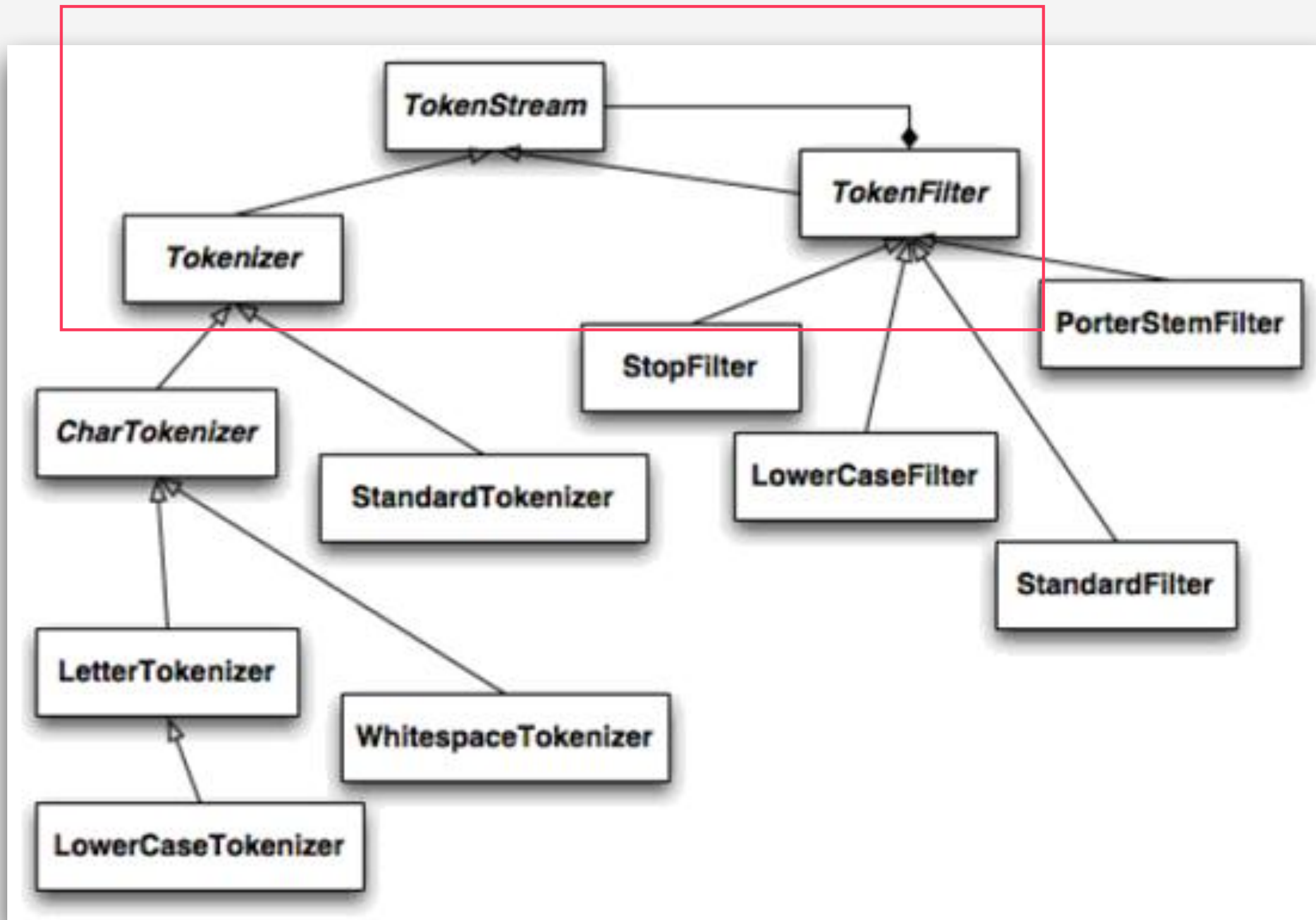
- Text Analysis는 색인어를 추출하는 과정
- Text Analysis는 문장을 Term들로 변환하는 것
- Term = “필드명 + Token”
- Token은 색인어 추출, 소문자 치환, 불용어 제거 등의 과정을 거쳐 텍스트로부터 쪼개진 조각들



[Analyzer chain]

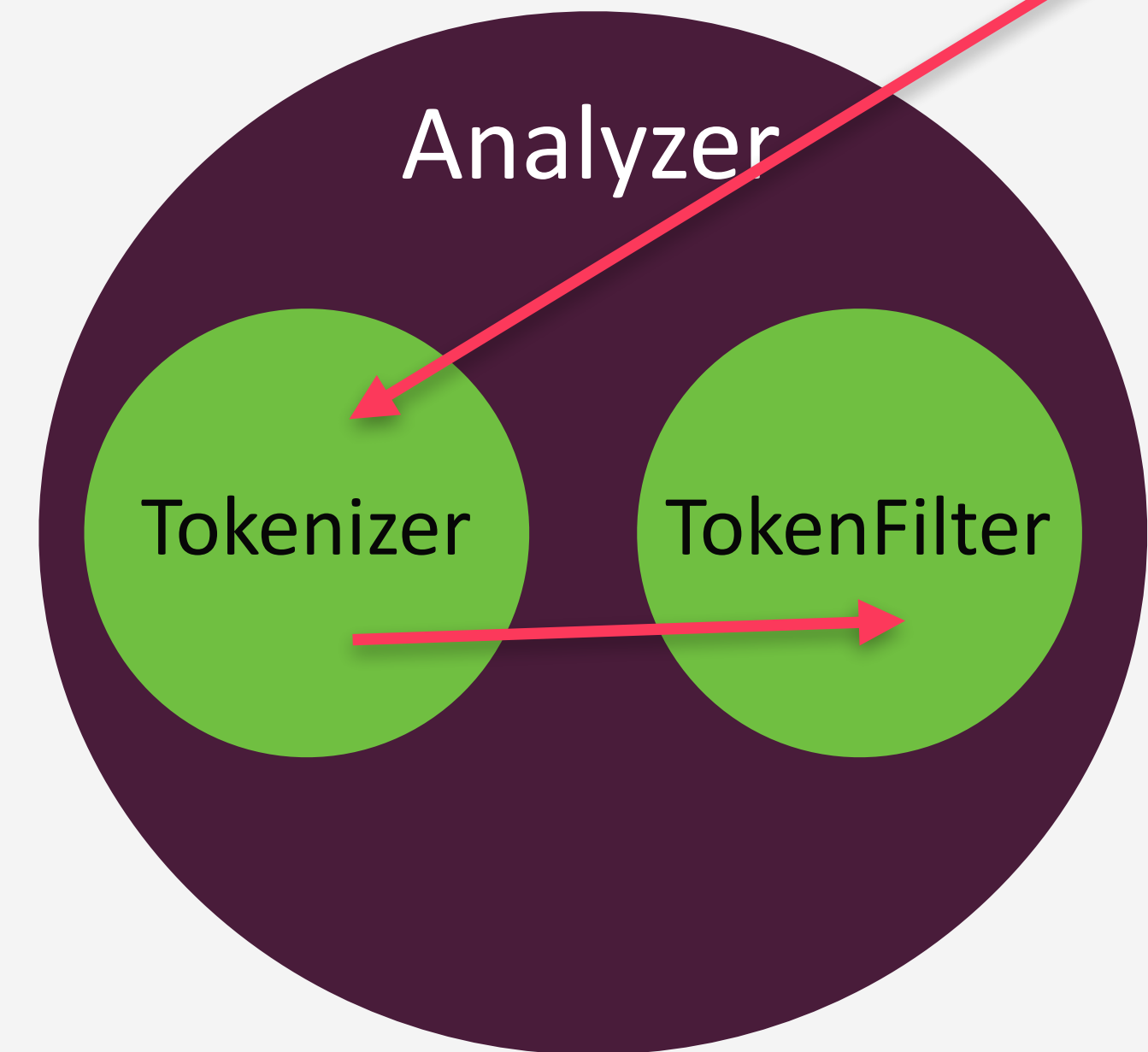
[Analysis Process]

루씬의 분석기 구조



루씬의 분석기 객체 구조

```
public TokenStream tokenStream(String fieldName, Reader reader)
```



루씬의 분석기 구조

TokenStream의 내부 구조

incrementToken () 메서드 : 토큰을 순차적으로 하나씩 읽음

CharTermAttribute : 문자 자체를 저장

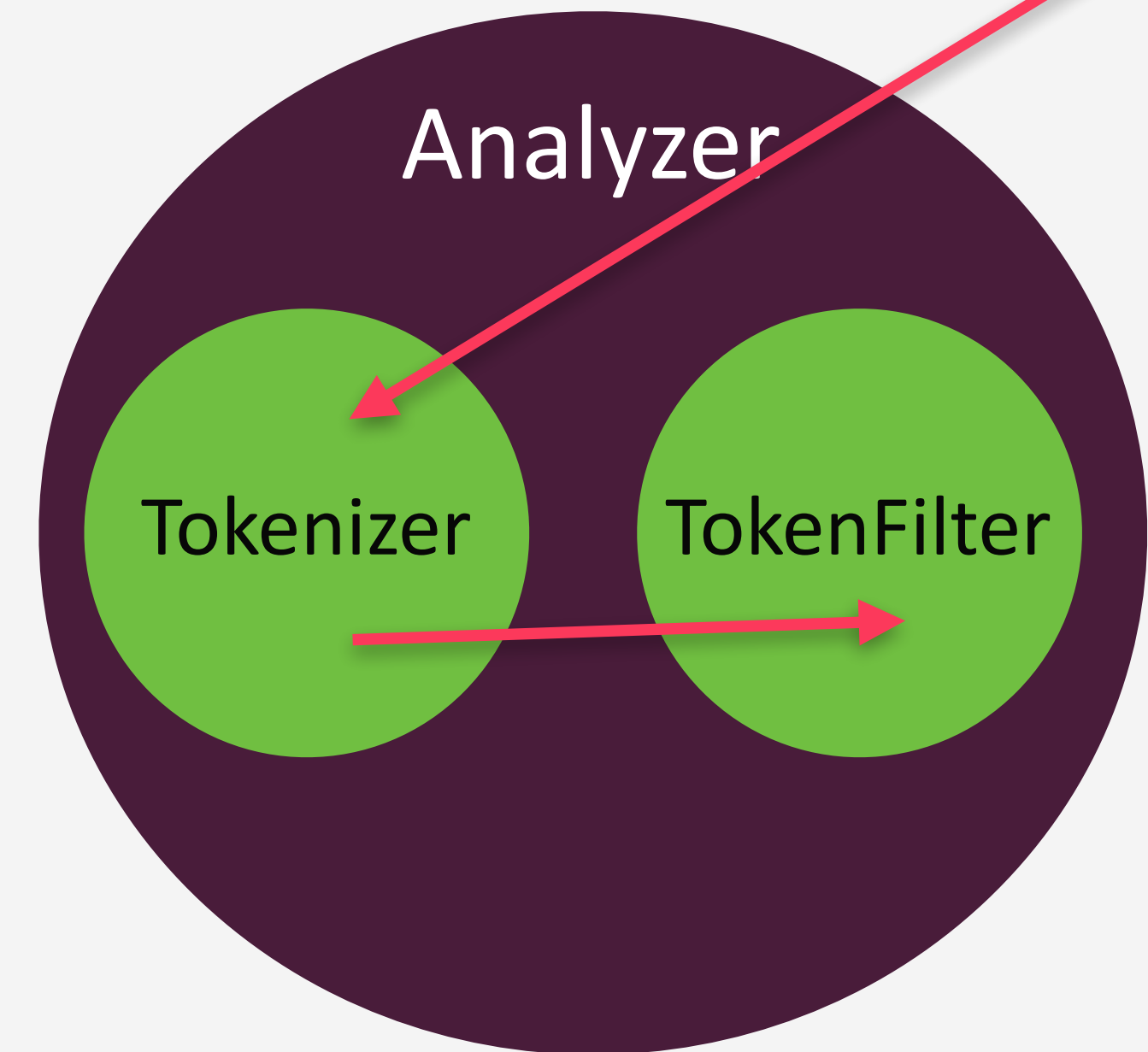
PositionIncrementAttribute : 포지션 증가값 저장

OffsetAttribute : 오프셋값 저장

TypeAttribute : 토큰의 속성값 저장

lucene 2.9 부터 속성기반 (재사용성)

```
public TokenStream tokenStream(String fieldName, Reader reader)
```



루씬의 분석기 객체 구조

루씬의 내장 분석기

<https://cwiki.apache.org/confluence/display/solr/Tokenizers>

- Standard Tokenizer
- Classic Tokenizer
- Keyword Tokenizer
- Letter Tokenizer
- Lower Case Tokenizer
- N-Gram Tokenizer
- Edge N-Gram Tokenizer
- ICU Tokenizer
- Path Hierarchy Tokenizer
- Regular Expression Pattern Tokenizer
- Simplified Regular Expression Pattern Tokenizer
- Simplified Regular Expression Pattern Splitting Tokenizer
- UAX29 URL Email Tokenizer
- White Space Tokenizer

<https://cwiki.apache.org/confluence/display/solr/Filter+Descriptions>

- ASCII Folding Filter
- Beider-Morse Filter
- Classic Filter
- Common Grams Filter
- Collation Key Filter
- Daitch-Mokotoff Soundex Filter
- Double Metaphone Filter
- Edge N-Gram Filter
- English Minimal Stem Filter
- English Possessive Filter
- Fingerprint Filter
- Flatten Graph Filter
- Hunspell Stem Filter
- Hyphenated Words Filter
- ICU Folding Filter
- ICU Normalizer 2 Filter
- ICU Transform Filter
- Keep Word Filter
- KStem Filter
- Length Filter
- Limit Token Count Filter
- Limit Token Offset Filter
- Limit Token Position Filter
- Lower Case Filter
- Managed Stop Filter
- Managed Synonym Filter
- N-Gram Filter
- Numeric Payload Token Filter
- Pattern Replace Filter
- Phonetic Filter
- Porter Stem Filter

루씬의 고급검색

필터 쿼리

검색할 때 검색 대상을 줄여주어 검색대상을 일부만 검색

결과내 검색 기능 및 조건 검색등에 사용

TermRangeFilter

지정한 범위에 속하는 텀만을 포함한 문서만 검색

NumericRangeFilter

지정한 범위의 숫자안에 포함된 문서만 검색

PrefixFilter

특정 필드에서 원하는 접두어를 갖고 있는 텀을 포함하는 문서만 검색

FieldCacheRangeFilter

특정 텀이나 숫자 범위에 포함된 문서만 결과에 포함
캐쉬사용

FieldCacheTermFilter

특정 텀이나 포함된 문서만 결과에 포함
캐쉬사용

PrefixFilter

특정 필드에서 원하는 접두어를 갖고 있는 텀을 포함하는 문서만 검색

필터를 사용하면 검색의 스코어에 영향을 주지 않으며 캐싱을 이용할수 있음

루씬의 캐시

검색시 특정 필드 값을 빠르게 조회할 필요 있을시 사용

1. 검색결과를 임시로 저장하여 동일 쿼리가 들어오면 검색하지 않고 캐쉬의 결과값을 리턴하는 방법
2. 많이 검색되는 키워드들에 대한 인덱싱을 빠른검색이 가능하게 별도로 관리

루씬은 Low Level에서 FieldCache 제공(5.0 이후 Deprecated)
현재 버전에서 QueryCache 존재

Solr의 경우

solr.search.LRUCache , solr.search.FastLRUCache, 및 solr.search.LFUCache 구현체를 이용

DocumentCache, FilterCache, QueryCache 기능 구현

Elastic의 경우

Node Query Cache

Shard Query Cache

FilterCache 등의 기능 구현

Geo Spatial Search

Spatial4J: Java 라이브러리는 지형 공간 모양

Lucene Spatial은 Spatial4J를 사용하여 문서와 함께 색인화 된 공간 객체를 생성

Spatial4J는 다각형 모양으로 작업에 적합

Lucene의 4.x 이후 추가

SpatialContext : spatial4j에서 제공되는 공간 객체 모델을 생성

SpatialStrategy : 일반적으로 RecursivePrefixTree

오타 교정 (Spellchecker)

Levenshtein edit distance

어떤 문자열을 삽입,삭제,변경을 몇번이나 해서 바꿀수 있는가 계산
그 최소값을 구해 유사도 판단의 척도 사용

Jaro–Winkler distance

문자열 s_1 과 s_2 사이의 Jaro distance d_j 의 정의

$$d_j = \begin{cases} 0 & \text{if } m = 0 \\ \frac{1}{3} \left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m} \right) & \text{otherwise} \end{cases}$$

m 은 두 문자열에서 공통적으로 존재하는 문자의 수
 t 는 두 문자열에 공통적으로 존재하나, 인덱스(순서)가 다른 문자의 수의 반

Jaro–Winkler distance d_w

$$d_w = d_j + (\ell p(1 - d_j))$$

p 는 고정된 scaling actor

공통 접두사가 있을경우 점수를 상향조정하기위해 사용하는 상수

p 는 0.25를 넘으면 안된다(넘으면 거리값이 1을 초과할수도 있음)

Winkler's work 에서 표준적인 값은 $p = 0.1$ 이다

ℓ 문자열의 공통 접두사의 길이로 최대 4개

주어진 문자열 s_1 MARTHA 와 s_2 MARHTA 가 있을 때:

- $m = 6$
- $|s_1| = 6$
- $|s_2| = 6$
- s_1 T/H 와 s_2 의 H/T가 공통적으로 존재하나 순서가 다른 문자이므로, $t = \frac{2}{2} = 1$

따라서 Jaro score 는:

$$d_j = \frac{1}{3} \left(\frac{6}{6} + \frac{6}{6} + \frac{6-1}{6} \right) = 0.944$$

Jaro-Winkler score 를 계산하기 위해 $p = 0.1$ 로 설정하고, 공통 접두사는 MAR 이므로:

- $\ell = 3$

따라서:

$$d_w = 0.944 + (3 * 0.1(1 - 0.944)) = 0.961$$

한글에서의 오타 교정 (Spellchecker)

한글의 음절을 자소단위로 분해 후 재결합하여 사용함 (정준 분해한 뒤 다시 정준 결합)

색인과 검색시

ICUNormalizer2Filter를 이용하여 방식으로 nfc 분해

출력 결과물 사용시

Normalizer.normalize({결과스트링}, Normalizer.Form.NFC)

<http://d2.naver.com/helloworld/76650>

유니코드 정규화 방법 참조

기타 고급 검색들

MultiSearcher (다중검색)

텀벡터를 활용한 MoreLikeThis

하일라이팅

감사합니다.

다음 주제 : 5일차 Elasticsearch Chapter 1