ISSN 2278-3083

Volume 4, No.2, March - April 2015

International Journal of Science and Applied Information Technology

Available Online at http://www.warse.org/ijsait/static/pdf/file/ijsait02422015.pdf

A Search Engine built using Open Source Software Technologies

Hye-Kyung Yang[1], **Minsoo Lee**[2]

[1]Department of Computer Science and Engineering, Ewha Womans University, Korea, yang88710@naver.com

[2]Department of Computer Science and Engineering, Ewha Womans University, Korea, mlee@ewha.ac.kr

ABSTRACT

In this project, we propose simple search engine for good restaurant based on Lucene. The project goal is to recommend that we want to find some restaurant information, such as restaurant name and food name. Using this search engine, we do not have to research restaurant information in web-site. We can easily find restaurant information using this search engine. The search engine was developed by JSP and Java. The search engine was constructed in the following way. First, we can create index on restaurant information. Before creating index, we should collect restaurant information from website. We will explain how to obtain these data. After that, we can start searching information that we want to find. We show results in the web page format.

Key words : engine, index, Lucene, restaurant, search,

1. INTRODUCTION

Lucene is open source information retrieval software library [1]. It is supported by the Apache Software Foundation and is released under the Apache Software License. Lucene has been ported to other programming languages including Delphi, Perl, C#, C++, Python, Ruby, and PHP [1]. Therefore, we use to develop simple engine for good restaurant based on Lucene. The project goal is to recommend that we want to find some restaurant information, such as restaurant name and food name. Using this search engine, we do not have to research restaurant information in web-site. We can easily find restaurant information using this search engine. The search engine was developed by Java based program. The search engine was constructed in the following way. First, we can create index on restaurant information. Before creating index, we should collect restaurant information. The restaurant information can be obtained from website. We will explain how to obtain information from website. After that, we created index with the model according to the text of parsing data. Create all possible terms in the index which are searched by network users as well as help people to manage and order extensive information and enable network users to quickly and easily retrieve any information they need. After indexing the documents, we can start searching information we need. The system shows to present results in the web page format.

The rest of the paper is organized as follows. Section 2 presents system architecture. Section 3 presents web parsing method. Section 4 presents creating index and Section 5 presents searching index. Section 6 concludes the paper.

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(No. 2012R1A1A2006850).

2. SYSTEM ARCHITECTURE

This section is system architecture part. The part presents to develop the simple search engine structure. Figure 1 shows the search engine structure.

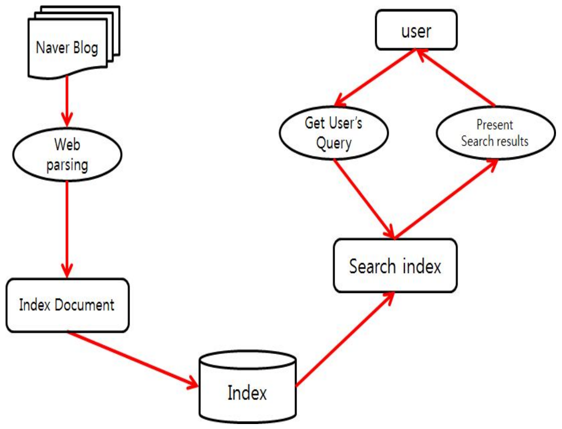


Figure 1: System Structure

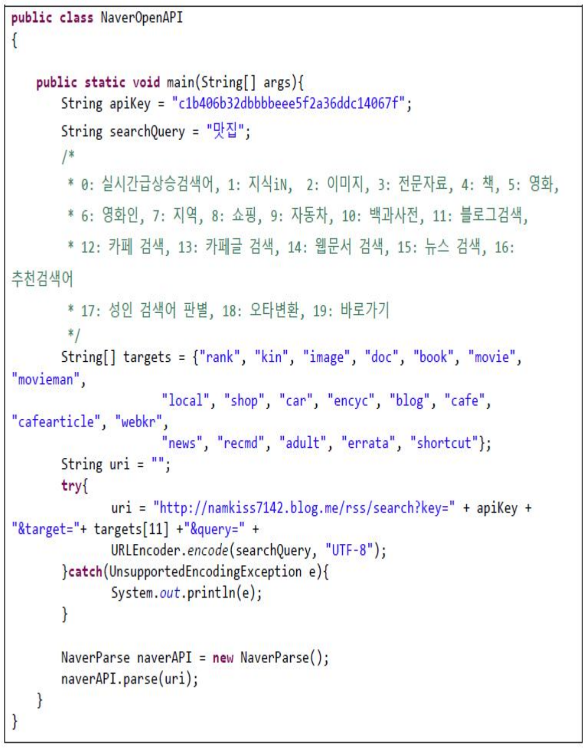
First, we have to collect good restaurants information. The method used web parsing. After that, we can create index with the model according to the text of parsing data. After indexing the documents, users can start to search information users need. Search requests are submitted by the users and information retrieval systems to preprocess and search the information eventually return user the information.

In this project, Lucene Development kit version is Lucene-4.7.8, it can also require java runtime environment above JDK1.6 version and need to import JAR package in to Eclipse.

3. WEB PARSING

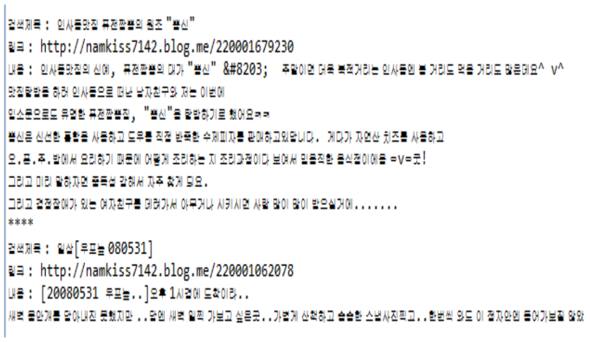
Before creating index, we should collect restaurant data, such as restaurant name, location and some information. Therefore, we can get web data using web crawler or web parsing. We got data from Naver blogs. However, Naver does not allow web robots so that we tried to use web parsing. Furthermore Naver provides searching open API, we can easily do web parsing.

Figure 2 shows NaverOpenAPI[2] Code. We can get blog data using these codes. Figure 3 shows parsing results. The good restaurant data collected total 381 documents.



```
public class NaverOpenAPI
{

    public static void main(String[] args){
        String apiKey = "c1b406b32dbbbbeee5f2a36ddc14067f";
        String searchQuery = "맛집";
        /*
         * 0: 실시간급상승검색어, 1: 지식iN,  2: 이미지, 3: 전문자료, 4: 책, 5: 영화,
         * 6: 영화인, 7: 지역, 8: 쇼핑, 9: 자동차, 10: 백과사전, 11: 블로그검색,
         * 12: 카페 검색, 13: 카페글 검색, 14: 웹문서 검색, 15: 뉴스 검색, 16:
추천검색어
         * 17: 성인 검색어 판별, 18: 오타변환, 19: 바로가기
         */
        String[] targets = {"rank", "kin", "image", "doc", "book", "movie",
"movieman",
                        "local", "shop", "car", "encyc", "blog", "cafe",
"cafearticle", "webkr",
                        "news", "recmd", "adult", "errata", "shortcut"};
        String uri = "";
        try{
                uri = "http://namkiss7142.blog.me/rss/search?key=" + apiKey +
"&target="+ targets[11] +"&query=" +
                URLEncoder.encode(searchQuery, "UTF-8");
        }catch(UnsupportedEncodingException e){
                System.out.println(e);
        }

        NaverParse naverAPI = new NaverParse();
        naverAPI.parse(uri);
    }
}
```

Figure 2: NaverOpenAPI Code



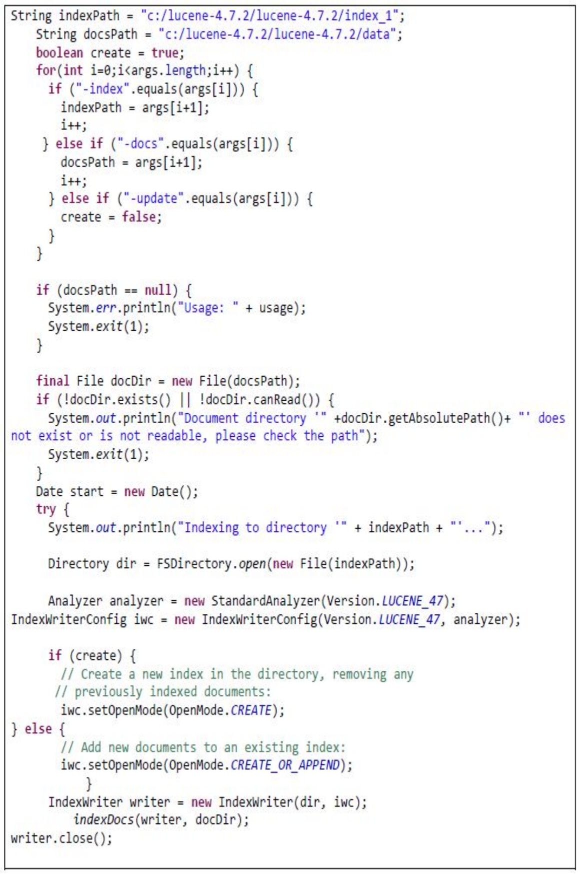
```
검색제목 : 인사동맛집 퓨전짬뽕의 원조 "뽕신"
링크 : http://namkiss7142.blog.me/220001679230
내용 : 인사동맛집의 신예, 퓨전짬뽕의 대가 "뽕신" &#8203;   주말이면 더욱 북적거리는 인사동엔 볼 거리도 먹을 거리도 많은데요^ v^
맛집탐방을 하러 인사동으로 떠난 남자친구와 저는 이번에
입소문으로도 유명한 퓨전짬뽕집, "뽕신"을 탐방하기로 했어요ㅋㅋ
뽕신은 신선한 홍합을 사용하고 도우를 직접 반죽한 수제피자를 판매하고있답니다. 게다가 자연산 치즈를 사용하고
오.픈.주.방에서 요리하기 때문에 어떻게 조리하는 지 조리과정이다 보여서 믿음직한 음식점이예요 =v=굿!
그리고 미리 말하자면 중독성 강해서 자주 찾게 되요.
그리고 결정장애가 있는 여자친구를 데려가서 아무거나 시키시면 사랑 많이 많이 받으실거예.......
****
검색제목 : 일상[우프놀 080531]
링크 : http://namkiss7142.blog.me/220001062078
내용 : [20080531 우프놀..]오후 1시경에 도착이라..
새벽 물안개를 담아내진 못했지만 ..담엔 새벽 일찍 가보고 싶은곳..가볍게 산책하고 슬슬한 스냅사진찍고..한번씩 와도 이 정자안엔 들어가보질 않았
```

Figure 3: parsing results

4. CREATE INDEX

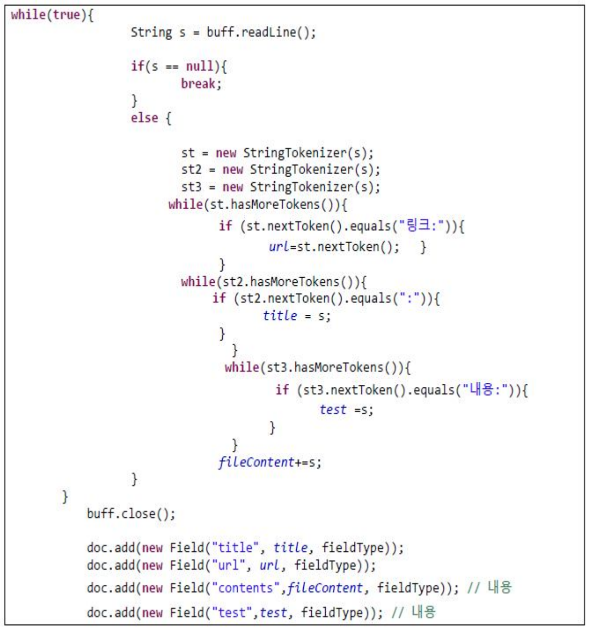
This section explains how to create indexing. Indexing can greatly improve the speed of information retrieval. In Lucene, an index is composed of segments, a segment is made up of documents, a document is composed of fields, and many terms consist of a field. The index process of Lucene is started from add Document method of IndexWriter. In Figure 4, show some creating index codes. In the API of Lucene, the main role of IndexWriter is to add documents to the indexing which provides us with the main interface for indexing.

First of all, creating indexing IndexWriter object which used StandardAnlyzer as an analysis tool, in order to generate indexing and store it into directory. We should separate indexing for fields in the documents, because collected document is TEXT file format. Figure 5 show separated indexing for fields and then stored terms of fields.



```
String indexPath = "c:/lucene-4.7.2/lucene-4.7.2/index_1";
    String docsPath = "c:/lucene-4.7.2/lucene-4.7.2/data";
    boolean create = true;
    for(int i=0;i<args.length;i++) {
      if ("-index".equals(args[i])) {
        indexPath = args[i+1];
        i++;
     } else if ("-docs".equals(args[i])) {
        docsPath = args[i+1];
        i++;
      } else if ("-update".equals(args[i])) {
        create = false;
      }
    }

    if (docsPath == null) {
      System.err.println("Usage: " + usage);
      System.exit(1);
    }

    final File docDir = new File(docsPath);
    if (!docDir.exists() || !docDir.canRead()) {
      System.out.println("Document directory '" +docDir.getAbsolutePath()+ "' does
not exist or is not readable, please check the path");
      System.exit(1);
    }
    Date start = new Date();
    try {
      System.out.println("Indexing to directory '" + indexPath + "'...");

      Directory dir = FSDirectory.open(new File(indexPath));

      Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_47);
IndexWriterConfig iwc = new IndexWriterConfig(Version.LUCENE_47, analyzer);

      if (create) {
        // Create a new index in the directory, removing any
       // previously indexed documents:
        iwc.setOpenMode(OpenMode.CREATE);
} else {
        // Add new documents to an existing index:
        iwc.setOpenMode(OpenMode.CREATE_OR_APPEND);
            }
      IndexWriter writer = new IndexWriter(dir, iwc);
         indexDocs(writer, docDir);
writer.close();
```

Figure 4: Create indexing code



```
while(true){
                String s = buff.readLine();

                if(s == null){
                        break;
                }
                else {

                        st = new StringTokenizer(s);
                        st2 = new StringTokenizer(s);
                        st3 = new StringTokenizer(s);
                      while(st.hasMoreTokens()){
                              if (st.nextToken().equals("링크:")){
                                      url=st.nextToken();   }
                              }
                        while(st2.hasMoreTokens()){
                             if (st2.nextToken().equals(":")){
                                     title = s;
                              }
                                }
                          while(st3.hasMoreTokens()){
                                  if (st3.nextToken().equals("내용:")){
                                          test =s;
                                  }
                          }
                      fileContent+=s;
                }
        }
           buff.close();

           doc.add(new Field("title", title, fieldType));
           doc.add(new Field("url", url, fieldType));
           doc.add(new Field("contents",fileContent, fieldType)); // 내용
           doc.add(new Field("test",test, fieldType)); // 내용
```

Figure 5: Store terms of fields

We tried to use StringTokenizer, that allows an application to break a string into tokens [3]. If the term meet a word "링크(link)", the next string data will be stored each field. The method is we can easily show page of results.

5. SEARCH INDEX

After indexing, the system established a search class. As we mentioned previously, we show search results in the web page format, as shown in Figure 6. The search pages were developed by JSP.

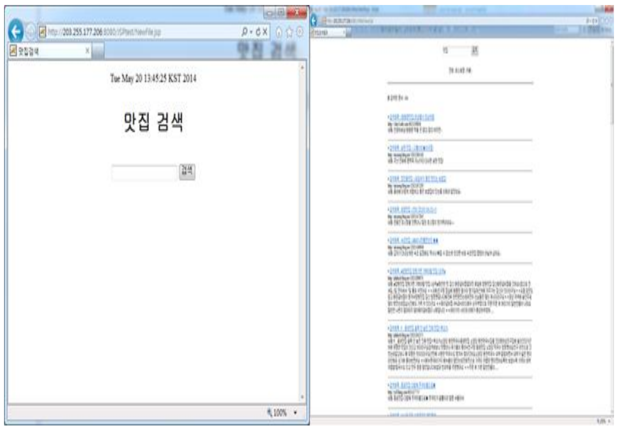
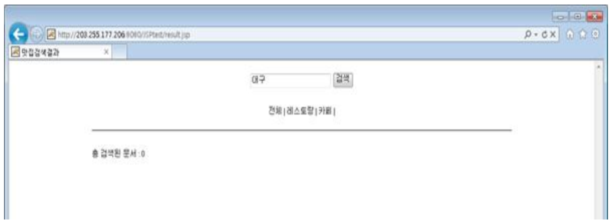


Figure 6: search pages

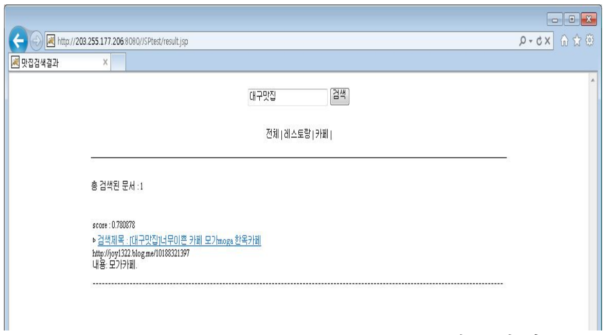
First, we tried to use basic searching module of Lucene. However, the basic search module has some problems. These are missing data and ranking. In the first problem of missing data, for example, if we input keyword is “대구(Daegu)”, we can’t get any results, as shown in Figure 7. However, Look at this Figure 8. Collected dataset has document including this keyword which has “대구맛집”. In this searching module think “대구” and “대구맛집” is different keywords.

Therefore, we should use WildCardQuery method. Supported wildcards are ‘*’, which matches any character sequence (including the empty one) and ‘?’, which matches any single character. ‘\’ is the escape character [4]. In other words, wildcards look like SQL ‘like’ query. The codes are as follows Figure 9.



총 검색된 문서 : 0

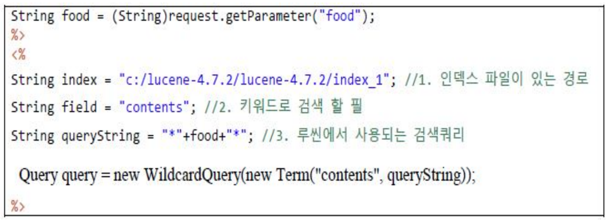
Figure 7: missing results



총 검색된 문서 : 1

score : 0.780878
▹ 검색제목 : [대구맛집]너무이쁜 카페 모가moga 한옥카페
http://joy1322.blog.me/10188321397
내용: 모가카페.

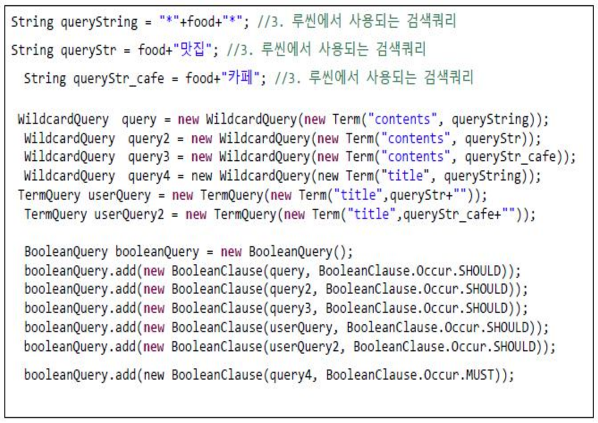
Figure 8: search results - keyword is “대구맛집 ”



```
String food = (String)request.getParameter("food");
%>
<%
String index = "c:/lucene-4.7.2/lucene-4.7.2/index_1"; //1. 인덱스 파일이 있는 경로
String field = "contents"; //2. 키워드로 검색 할 필
String queryString = "*"+food+"*"; //3. 루씬에서 사용되는 검색쿼리

 Query query = new WildcardQuery(new Term("contents", queryString));

%>
```

Figure 9: WildCardquery code

However, the method also has a problem. When using WildcardQuery, the score is always 1.0. Because of this problem, the lowest relevant document can be located to high rank. We solved the problem using BooleanQuery with WildcardQuery. The codes are as follows Figure 10.

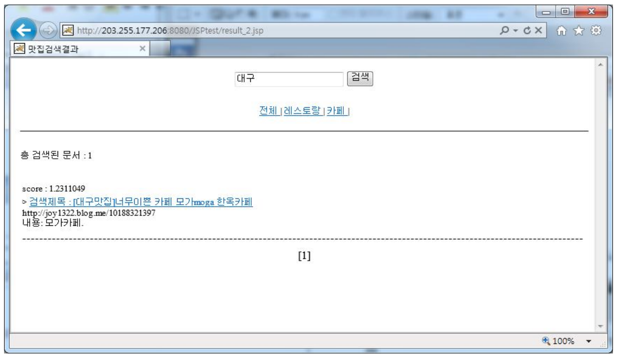


```
String queryString = "*"+food+"*"; //3. 루씬에서 사용되는 검색쿼리
String queryStr = food+"맛집"; //3. 루씬에서 사용되는 검색쿼리
 String queryStr_cafe = food+"카페"; //3. 루씬에서 사용되는 검색쿼리

WildcardQuery  query = new WildcardQuery(new Term("contents", queryString));
 WildcardQuery  query2 = new WildcardQuery(new Term("contents", queryStr));
 WildcardQuery  query3 = new WildcardQuery(new Term("contents", queryStr_cafe));
 WildcardQuery  query4 = new WildcardQuery(new Term("title", queryString));
TermQuery userQuery = new TermQuery(new Term("title",queryStr+""));
 TermQuery userQuery2 = new TermQuery(new Term("title",queryStr_cafe+""));

 BooleanQuery booleanQuery = new BooleanQuery();
 booleanQuery.add(new BooleanClause(query, BooleanClause.Occur.SHOULD));
 booleanQuery.add(new BooleanClause(query2, BooleanClause.Occur.SHOULD));
 booleanQuery.add(new BooleanClause(query3, BooleanClause.Occur.SHOULD));
 booleanQuery.add(new BooleanClause(userQuery, BooleanClause.Occur.SHOULD));
 booleanQuery.add(new BooleanClause(userQuery2, BooleanClause.Occur.SHOULD));

 booleanQuery.add(new BooleanClause(query4, BooleanClause.Occur.MUST));
```

Figure 10: BooleanQuery with WildcardQuery codes

The codes contained WildcardQuery,TermQuery and BooleanQuery. This method is a query that matches document matching Boolean combination of other queries, e.g TermQuery and WildcardQuery and other queries. BooleanClause.Occuer.SHOULD operator indicates that should appear in the matching documents [5].

BooleanClause.Occuer.MUST operator indicates that must appear in the matching documents [5]. In this project, we want to find more correct results. Thus, the keyword has to contain the title of document. Figure 11 shows the results pages for input keyword.



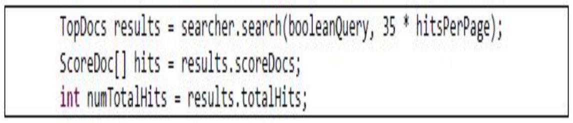
총 검색된 문서 : 1

score : 1.2311049
▹ 검색제목 : [대구맛집]너무이쁜 카페 모가moga 한옥카페
http://joy1322.blog.me/10188321397
내용: 모가카페.

[1]

Figure 11: Results page for input keyword

In this project, we used scoring method, as shown the Figure 12. The code usually uses to calculate document score in the Lucene. We got each of document score using this scoring method.



```
TopDocs results = searcher.search(booleanQuery, 35 * hitsPerPage);
ScoreDoc[] hits = results.scoreDocs;
int numTotalHits = results.totalHits;
```

Figure 12: Score code

6. CONCLUSION

This section explains how to create indexing. Indexing can greatly improve the speed of information retrieval. In Lucene, an index is composed of segments, a segment is made up of documents, a document is composed of fields, and many terms consist of a field. The index process of Lucene is started from add Document method of IndexWriter. In Figure 4, show some creating index codes. In the API of Lucene, the main role of IndexWriter is to add documents to the indexing which provides us with the main interface for indexing.

REFERENCES

1. **Lucene** wikipedia , http://en.wikipedia.org/wiki/Lucene
2. **Naver Open API**, http://developer.naver.com/wiki/pages/OpenAPI
3. **JAVA Stringtokenizer**, http://www.tutorialspoint.com/java/util/java_util_stringtokenizer.htm
4. **Lucene WildCardQuery**, https://lucene.apache.org/core/4_0_0/core/org/apache/lucene/search/WildcardQuery.html
5. **Lucene BooleanQuery**, http://lucene.apache.org/core/3_0_3/api/all/org/apache/lucene/search/BooleanClause.Occur.html