Architecture of Enterprise Applications 8
Full-text Searching

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
http://reins.se.sjtu.edu.cn/~chenhp
e-mail: chen-hp@sjtu.edu.cn

Contents and Objectives

- Contents
  - Lucene
  - Solr
  - Elasticsearch

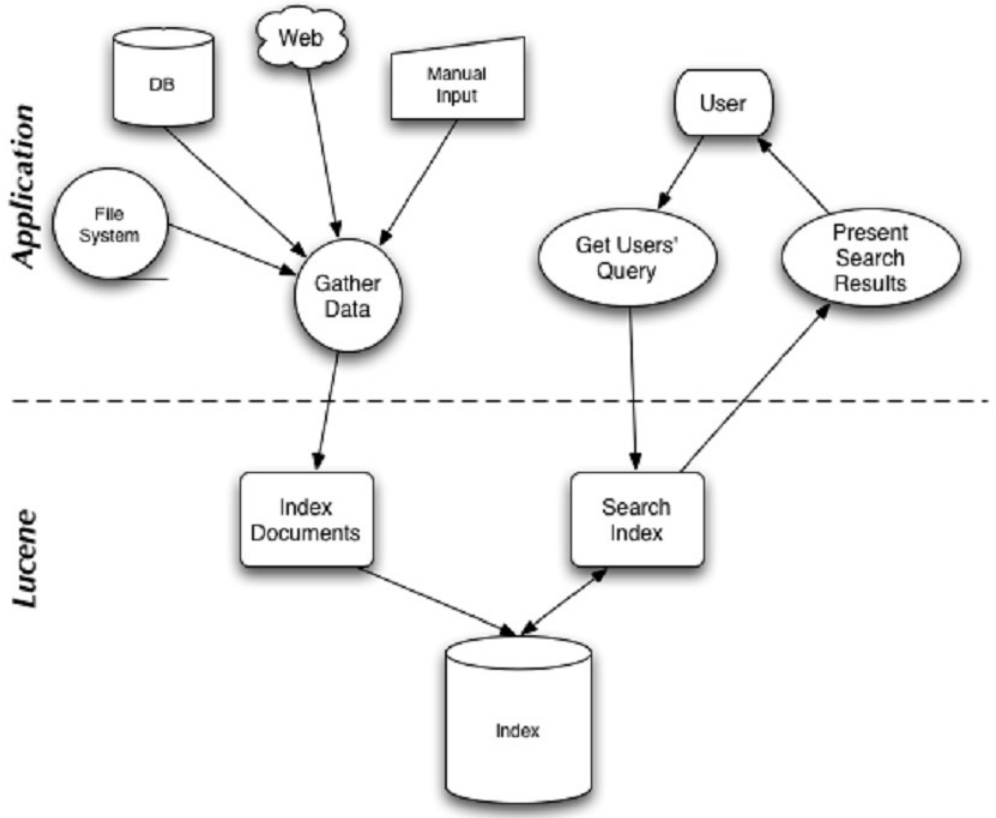
- Objectives
  - 能够根据业务需求，识别适合全文搜索引擎的场景，设计并实现基于搜索引擎的全文搜索方案

Lucene

- Lucene is a high performance, scalable Information Retrieval (IR) library.
  - It lets you add indexing and searching capabilities to your applications.
  - Lucene is a mature, free, open-source project implemented in Java.
  - it’s a member of the popular Apache Jakarta family of projects, licensed under the liberal Apache Software License.

- Lucene provides a simple yet powerful core API
  - that requires minimal understanding of full-text indexing and searching.

Lucene


Application
Lucene
DB
Web
Manual Input
File System
Gather Data
User
Get Users' Query
Present Search Results
Index Documents
Search Index
Index

Indexing

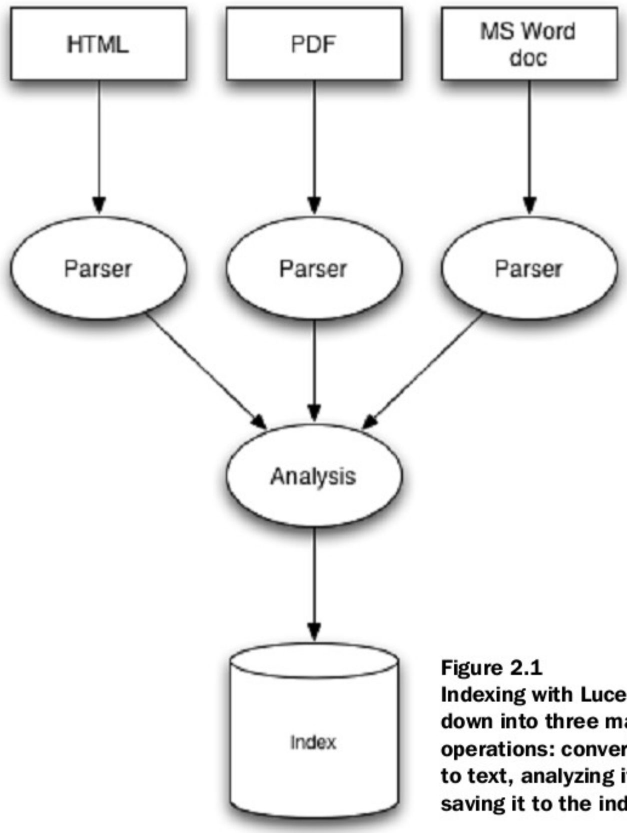


Figure 2.1
Indexing with Lucene breaks down into three main operations: converting data to text, analyzing it, and saving it to the index.

indexing

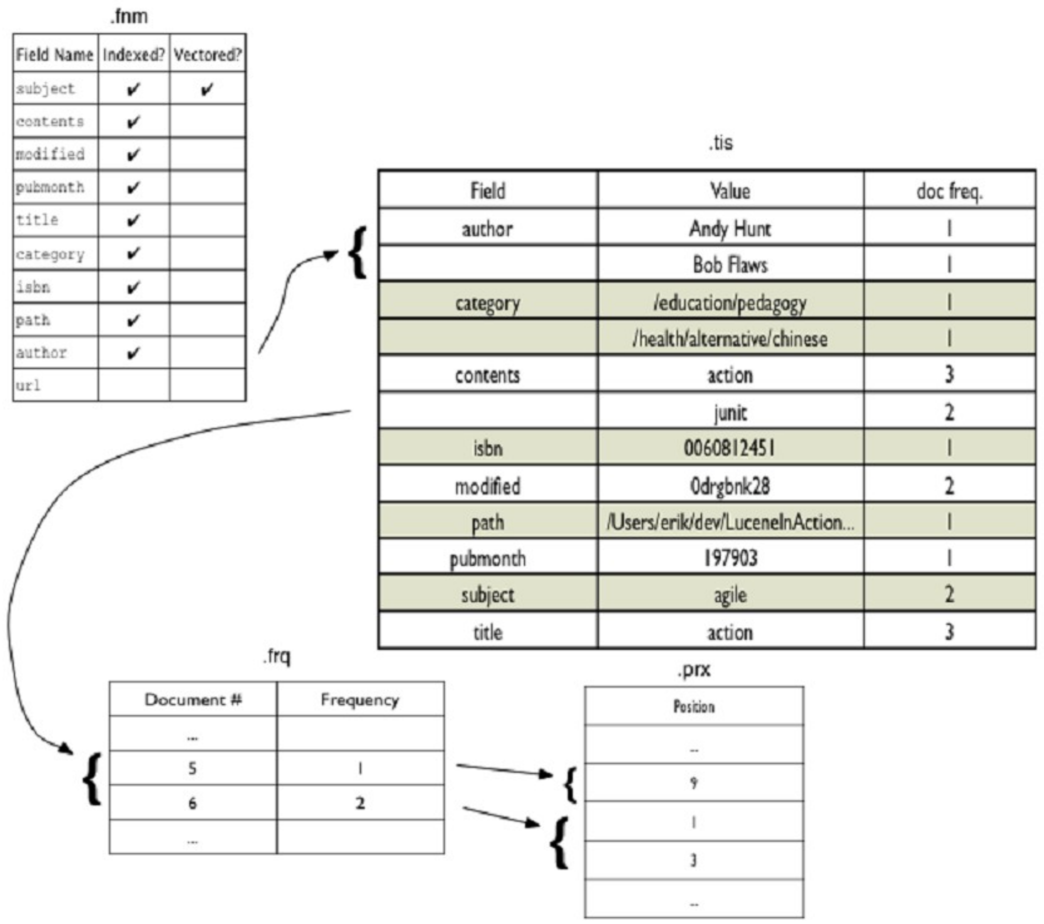
- At the heart of all search engines is the concept of indexing:
  - processing the original data into a highly efficient cross-reference lookup in order to facilitate rapid searching.

- Suppose you needed to search a large number of files, and you wanted to be able to find files that contained a certain word or a phrase
  - A naïve approach would be to sequentially scan each file for the given word or phrase.
  - This approach has a number of flaws, the most obvious of which is that it doesn’t scale to larger file sets or cases where files are very large.

indexing

- This is where indexing comes in:
  - To search large amounts of text quickly, you must first index that text and convert it into a format that will let you search it rapidly, eliminating the slow sequential scanning process.
  - This conversion process is called indexing, and its output is called an index.
  - You can think of an index as a data structure that allows fast random access to words stored inside it.

Inverting index



.fnm

Field Name	Indexed?	Vectored?
subject	✔	✔
contents	✔	
modified	✔	
pubmonth	✔	
title	✔	
category	✔	
isbn	✔	
path	✔	
author	✔	
url		

.tis

Field	Value	doc freq.
author	Andy Hunt	1
	Bob Flaws	1
category	/education/pedagogy	1
	/health/alternative/chinese	1
contents	action	3
	junit	2
isbn	0060812451	1
modified	0drgbnk28	2
path	/Users/erik/dev/LuceneInAction...	1
pubmonth	197903	1
subject	agile	2
title	action	3

.frq

Document #	Frequency
...	
5	1
6	2
...	

.prx

Position
...
9
1
3
...

Searching

- Searching is the process of looking up words in an index to find documents where they appear.

- The quality of a search is typically described using precision and recall metrics.
  - Recall measures how well the search system finds relevant documents, whereas precision measures how well the system filters out the irrelevant documents.

- A number of other factors
  - speed and the ability to quickly search large quantities of text.
  - Support for single and multi term queries, phrase queries, wildcards, result ranking, and sorting are also important, as is a friendly syntax for entering those queries.

Core indexing classes

- The Lucene API is divided into several packages:
  - org.apache.lucene.analysis defines an abstract Analyzer API for converting text from a Reader into a TokenStream, an enumeration of token Attributes.
    - A TokenStream can be composed by applying TokenFilters to the output of a Tokenizer.
    - Tokenizers and TokenFilters are strung together and applied with an Analyzer.
    - analysis-common provides a number of Analyzer implementations, including StopAnalyzer and the grammar-based StandardAnalyzer.
  - org.apache.lucene.codecs provides an abstraction over the encoding and decoding of the inverted index structure, as well as different implementations that can be chosen depending upon application needs.
  - org.apache.lucene.document provides a simple Document class.
    - A Document is simply a set of named Fields, whose values may be strings or instances of Reader.

Core indexing classes

- The Lucene API is divided into several packages:
  - org.apache.lucene.index provides two primary classes:
    - IndexWriter, which creates and adds documents to indices; and IndexReader, which accesses the data in the index.
  - org.apache.lucene.search provides data structures to represent queries (ie TermQuery for individual words, PhraseQuery for phrases, and BooleanQuery for boolean combinations of queries) and the IndexSearcher which turns queries into TopDocs.
    - A number of QueryParsers are provided for producing query structures from strings or xml.
  - org.apache.lucene.store defines an abstract class for storing persistent data, the Directory, which is a collection of named files written by an IndexOutput and read by an IndexInput.
    - Multiple implementations are provided, but FSDirectory is generally recommended as it tries to use operating system disk buffer caches efficiently.
  - org.apache.lucene.util contains a few handy data structures and util classes, ie FixedBitSet and PriorityQueue.

A sample application

- To use Lucene, an application should:Create Documents by adding Fields;
  1. Create an IndexWriter and add documents to it with addDocument();
  2. Call QueryParser.parse() to build a query from a string; and
  3. Create an IndexSearcher and pass the query to its search() method.

Creating an Index

```java
/**
 * Index all text files under a directory.
 */
public class IndexFiles implements AutoCloseable {
public static void main(String[] args) throws Exception {
    String indexPath = "index";
    String docsPath = null;
    String vectorDictSource = null;
    boolean create = true;


    final Path docDir = Paths.get(docsPath);
    Date start = new Date();
    try {
        System.out.println("Indexing to directory '" + indexPath + "'...");

        Directory dir = FSDirectory.open(Paths.get(indexPath));
        Analyzer analyzer = new StandardAnalyzer();
        IndexWriterConfig iwc = new IndexWriterConfig(analyzer);

        if (create) {
            iwc.setOpenMode(OpenMode.CREATE);
        } else {
            iwc.setOpenMode(OpenMode.CREATE_OR_APPEND);
        }

        KnnVectorDict vectorDictInstance = null;
        long vectorDictSize = 0;
        if (vectorDictSource != null) {
            KnnVectorDict.build(Paths.get(vectorDictSource), dir, KNN_DICT);
            vectorDictInstance = new KnnVectorDict(dir, KNN_DICT);
            vectorDictSize = vectorDictInstance.ramBytesUsed();
        }
```

```java
        try (IndexWriter writer = new IndexWriter(dir, iwc); IndexFiles indexFiles = new IndexFiles(vectorDictInstance)) {
            indexFiles.indexDocs(writer, docDir);
        } finally {
            IOUtils.close(vectorDictInstance);
        }

        Date end = new Date();
        try (IndexReader reader = DirectoryReader.open(dir)) {
            System.out.println(
                    "Indexed "
                            + reader.numDocs()
                            + " documents in "
                            + (end.getTime() - start.getTime())
                            + " ms");
            if (reader.numDocs() > 100
                    && vectorDictSize < 1_000_000
                    && System.getProperty("smoketester") == null) {
                throw new RuntimeException(
                        "Are you (ab)using the toy vector dictionary? See the package javadocs to understand why you got this exception.");
            }
        }
    } catch (IOException e) {
        System.out.println(" caught a " + e.getClass() + "\n with message: " + e.getMessage());
    }
}
```

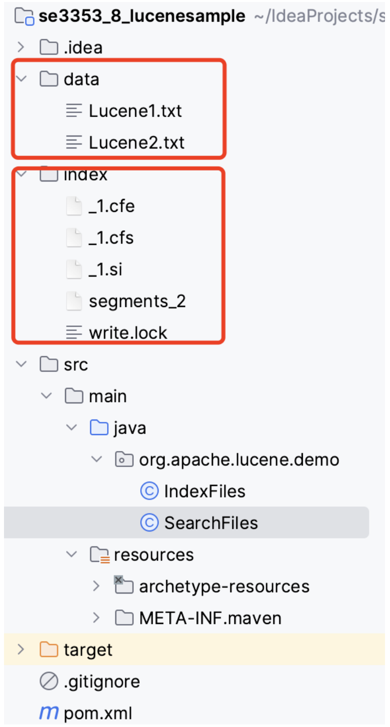
Creating an Index

```
/**
 * Indexes the given file using the given writer, or if a directory is given, recurses over files
 * and directories found under the given directory.
 */
void indexDocs(final IndexWriter writer, Path path) throws IOException {
    if (Files.isDirectory(path)) {
        Files.walkFileTree(
                path,
                new SimpleFileVisitor<>() {
                    @Override
                    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {
                        try {
                            indexDoc(writer, file, attrs.lastModifiedTime().toMillis());
                        } catch (
                                @SuppressWarnings("unused")
                                IOException ignore) {
                            ignore.printStackTrace(System.err);
                            // don't index files that can't be read.
                        }
                        return FileVisitResult.CONTINUE;
                    }
                });
    } else {
        indexDoc(writer, path, Files.getLastModifiedTime(path).toMillis());
    }
}
```

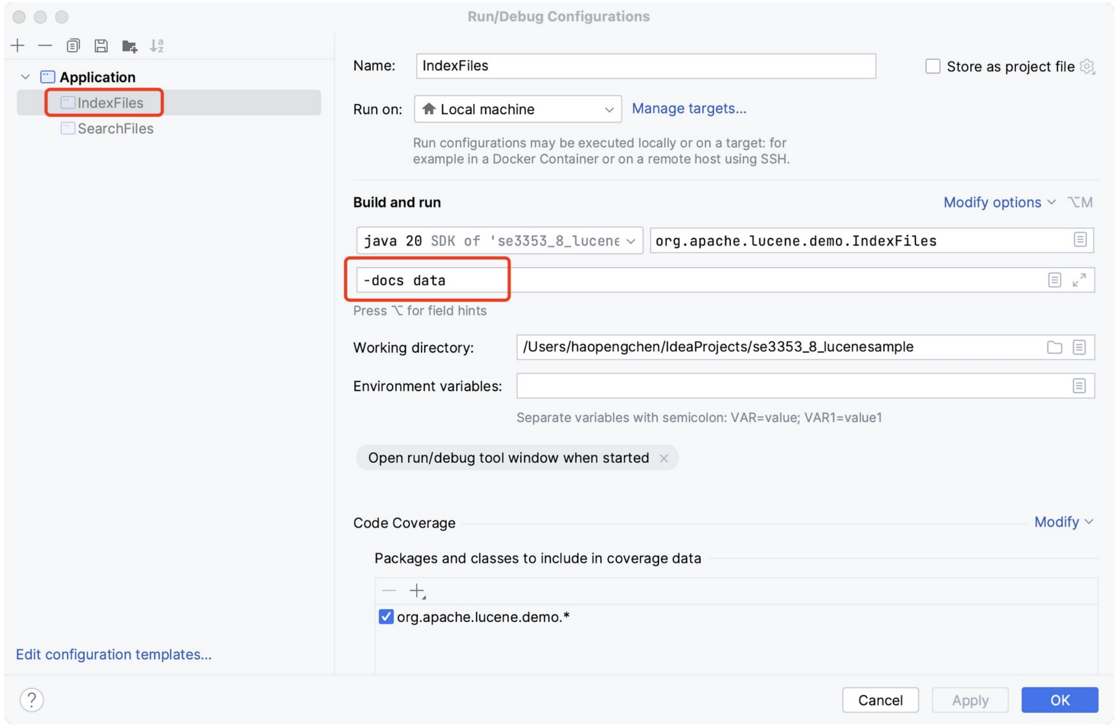
Creating an Index

```java
/**
 * Indexes a single document
 */
void indexDoc(IndexWriter writer, Path file, long lastModified) throws IOException {
    try (InputStream stream = Files.newInputStream(file)) {
        Document doc = new Document();

        doc.add(new KeywordField(“path”, file.toString(), Field.Store.YES));
        doc.add(new LongField(“modified”, lastModified, Field.Store.NO));
        doc.add(
                new TextField(
                        "contents",
                        new BufferedReader(new InputStreamReader(stream, StandardCharsets.UTF_8))));

        if (demoEmbeddings != null) {
            try (InputStream in = Files.newInputStream(file)) {
                float[] vector =
                        demoEmbeddings.computeEmbedding(
                                new BufferedReader(new InputStreamReader(in, StandardCharsets.UTF_8)));
                doc.add(
                        new KnnFloatVectorField(
                                "contents-vector", vector, VectorSimilarityFunction.DOT_PRODUCT));
            }
        }

        if (writer.getConfig().getOpenMode() == OpenMode.CREATE) {
            System.out.println("adding " + file);
            writer.addDocument(doc);
        } else {
            System.out.println("updating " + file);
            writer.updateDocument(new Term("path", file.toString()), doc);
        }
    }
}
```

Running Indexer



se3353_8_lucenesample ~/IdeaProjects/s
- .idea
- data
  - Lucene1.txt
  - Lucene2.txt
- index
  - _1.cfe
  - _1.cfs
  - _1.si
  - segments_2
  - write.lock
- src
  - main
    - java
      - org.apache.lucene.demo
        - IndexFiles
        - SearchFiles
    - resources
      - archetype-resources
      - META-INF.maven
- target
- .gitignore
- pom.xml



Run/Debug Configurations

- Application
  - IndexFiles
  - SearchFiles

Name: IndexFiles

Store as project file

Run on: Local machine — Manage targets...

Run configurations may be executed locally or on a target: for example in a Docker Container or on a remote host using SSH.

Build and run — Modify options ⌥M

```
java 20 SDK of 'se3353_8_lucene    org.apache.lucene.demo.IndexFiles
-docs data
```

Press ⌥ for field hints

Working directory: /Users/haopengchen/IdeaProjects/se3353_8_lucenesample

Environment variables:

Separate variables with semicolon: VAR=value; VAR1=value1

Open run/debug tool window when started

Code Coverage — Modify

Packages and classes to include in coverage data

✓ org.apache.lucene.demo.*

Edit configuration templates...

Searching an index

REliable, INtelligent & Scalable Systems

```
        // Simple command-line based search demo.
public class SearchFiles {
  public static void main(String[] args) throws Exception {

    String index = "index";
    String field = "contents";
    String queries = null;
    int repeat = 0;
    boolean raw = false;
    int knnVectors = 0;
    String queryString = null;
    int hitsPerPage = 10;


    DirectoryReader reader = DirectoryReader.open(FSDirectory.open(Paths.get(index)));
    IndexSearcher searcher = new IndexSearcher(reader);
    Analyzer analyzer = new StandardAnalyzer();
    KnnVectorDict vectorDict = null;
    if (knnVectors > 0) {
      vectorDict = new KnnVectorDict(reader.directory(), IndexFiles.KNN_DICT);
    }
    BufferedReader in;
    if (queries != null) {
      in = Files.newBufferedReader(Paths.get(queries), StandardCharsets.UTF_8);
    } else {
      in = new BufferedReader(new InputStreamReader(System.in, StandardCharsets.UTF_8));
    }
    QueryParser parser = new QueryParser(field, analyzer);
```

Searching an index

REliable, INtelligent & Scalable Systems

```
while (true) {
        if (queries == null && queryString == null) { // prompt the user
            System.out.println("Enter query: ");
        }

        String line = queryString != null ? queryString : in.readLine();

        if (line == null || line.length() == -1) {
            break;
        }

        line = line.trim();
        if (line.length() == 0) {
            break;
        }

        Query query = parser.parse(line);
        if (knnVectors > 0) {
            query = addSemanticQuery(query, vectorDict, knnVectors);
        }
        System.out.println("Searching for: " + query.toString(field));
```

```
        if (repeat > 0) { // repeat & time as benchmark
            Date start = new Date();
            for (int i = 0; i < repeat; i++) {
                searcher.search(query, 100);
            }
            Date end = new Date();
            System.out.println("Time: " + (end.getTime() - start.getTime()) + "ms");
        }

        doPagingSearch(in, searcher, query, hitsPerPage, raw, queries == null && queryString
== null);

        if (queryString != null) {
            break;
        }
    }
    if (vectorDict != null) {
        vectorDict.close();
    }
    reader.close();
}
```

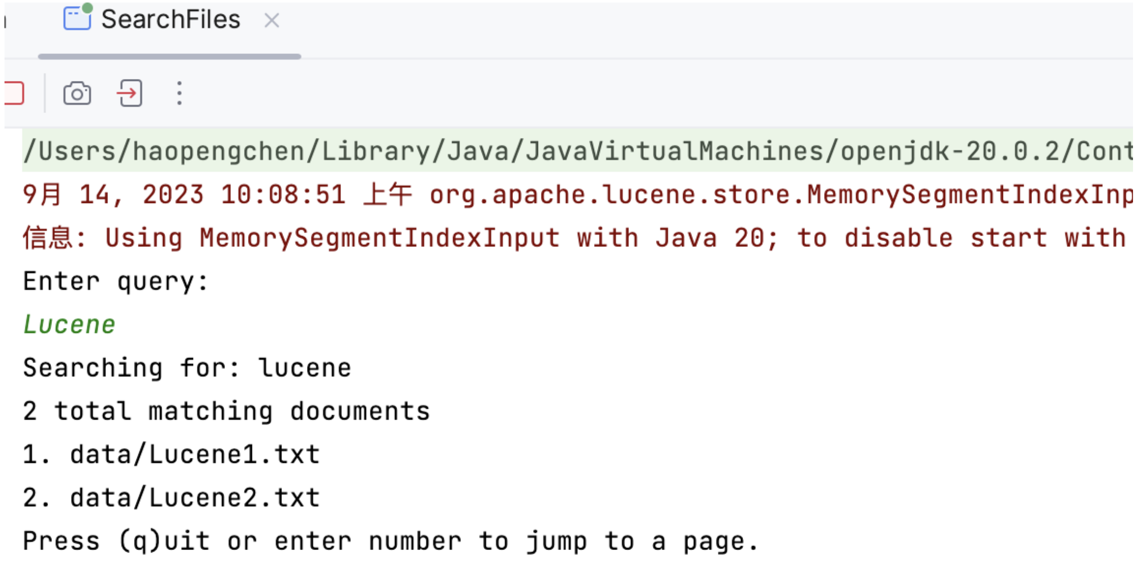
Searching an index

```
public static void doPagingSearch(
        BufferedReader in,
        IndexSearcher searcher,
        Query query,
        int hitsPerPage,
        boolean raw,
        boolean interactive)
        throws IOException {

    // Collect enough docs to show 5 pages
    TopDocs results = searcher.search(query, 5 * hitsPerPage);
    ScoreDoc[] hits = results.scoreDocs;

    int numTotalHits = Math.toIntExact(results.totalHits.value);
    System.out.println(numTotalHits + " total matching documents");

    int start = 0;
    int end = Math.min(numTotalHits, hitsPerPage);

    while (true) {
        if (end > hits.length) {
            System.out.println(
                    "Only results 1 - "
                            + hits.length
                            + " of "
                            + numTotalHits
                            + " total matching documents collected.");
            System.out.println("Collect more (y/n) ?");
            String line = in.readLine();
            if (line.length() == 0 || line.charAt(0) == 'n') {
                break;
            }

            hits = searcher.search(query, numTotalHits).scoreDocs;
        }

        end = Math.min(hits.length, start + hitsPerPage);
```

```
        StoredFields storedFields = searcher.storedFields();
        for (int i = start; i < end; i++) {
            if (raw) { // output raw format
                System.out.println("doc=" + hits[i].doc + " score=" + hits[i].score);
                continue;
            }

            Document doc = storedFields.document(hits[i].doc);
            String path = doc.get("path");
            if (path != null) {
                System.out.println((i + 1) + ". " + path);
                String title = doc.get("title");
                if (title != null) {
                    System.out.println("   Title: " + doc.get("title"));
                }
            } else {
                System.out.println((i + 1) + ". " + "No path for this document");
            }
        }

        if (!interactive || end == 0) {
            break;
        }
```

```
        if (numTotalHits >= end) {
            boolean quit = false;
            while (true) {
                System.out.print("Press ");
                if (start - hitsPerPage >= 0) {
                    System.out.print("(p)revious page, ");
                }
                if (start + hitsPerPage < numTotalHits) {
                    System.out.print("(n)ext page, ");
                }
                System.out.println("(q)uit or enter number to jump to a page.");

                String line = in.readLine();
                if (line.length() == 0 || line.charAt(0) == 'q') {
                    quit = true;
                    break;
                }
                if (line.charAt(0) == 'p') {
                    start = Math.max(0, start - hitsPerPage);
                    break;
                } else if (line.charAt(0) == 'n') {
                    if (start + hitsPerPage < numTotalHits) {
                        start += hitsPerPage;
                    }
                    break;
                } else {
                    int page = Integer.parseInt(line);
                    if ((page - 1) * hitsPerPage < numTotalHits) {
                        start = (page - 1) * hitsPerPage;
                        break;
                    } else {
                        System.out.println("No such page");
                    }
                }
            }
            if (quit) break;
            end = Math.min(numTotalHits, start + hitsPerPage);
        }
    }
}
```

Running Searcher



```
/Users/haopengchen/Library/Java/JavaVirtualMachines/openjdk-20.0.2/Cont
9月 14, 2023 10:08:51 上午 org.apache.lucene.store.MemorySegmentIndexInp
信息: Using MemorySegmentIndexInput with Java 20; to disable start with
Enter query:
Lucene
Searching for: lucene
2 total matching documents
1. data/Lucene1.txt
2. data/Lucene2.txt
Press (q)uit or enter number to jump to a page.
```

Index File Formats

- The fundamental concepts in Lucene are index, document, field and term.

- An index contains a sequence of documents.
  - A document is a sequence of fields.
  - A field is a named sequence of terms.
  - A term is a sequence of bytes.

- The same sequence of bytes in two different fields is considered a different term.
  - Thus terms are represented as a pair: the string naming the field, and the bytes within the field.

Index File Formats

- Inverted Indexing
  - Lucene's index stores terms and statistics about those terms in order to make term-based search more efficient.
  - Lucene's terms index falls into the family of indexes known as an inverted index.
  - This is because it can list, for a term, the documents that contain it.
  - This is the inverse of the natural relationship, in which documents list terms.

- Types of Fields
  - In Lucene, fields may be stored, in which case their text is stored in the index literally, in a non-inverted manner. Fields that are inverted are called indexed.
  - A field may be both stored and indexed.
  - The text of a field may be tokenized into terms to be indexed, or the text of a field may be used literally as a term to be indexed.
  - Most fields are tokenized, but sometimes it is useful for certain identifier fields to be indexed literally.

Types of Fields

- Expert: directly create a field for a document. Most users should use one of the sugar subclasses:
  - TextField: Reader or String indexed for full-text search
  - StringField: String indexed verbatim as a single token
  - IntPoint: int indexed for exact/range queries.
  - LongPoint: long indexed for exact/range queries.
  - FloatPoint: float indexed for exact/range queries.
  - DoublePoint: double indexed for exact/range queries.
  - SortedDocValuesField: byte[] indexed column-wise for sorting/faceting
  - SortedSetDocValuesField: SortedSet<byte[]> indexed column-wise for sorting/faceting
  - NumericDocValuesField: long indexed column-wise for sorting/faceting
  - SortedNumericDocValuesField: SortedSet<long> indexed column-wise for sorting/faceting
  - StoredField: Stored-only value for retrieving in summary results
- A field is a section of a Document.
  - Each field has three parts: name, type and value.
  - Values may be text (String, Reader or pre-analyzed TokenStream), binary (byte[]), or numeric (a Number).
  - Fields are optionally stored in the index, so that they may be returned with hits on the document.

Index File Formats

- Segments
  - Lucene indexes may be composed of multiple sub-indexes, or segments.
  - Each segment is a fully independent index, which could be searched separately.
  - Indexes evolve by:
    - Creating new segments for newly added documents.
    - Merging existing segments.
    - Searches may involve multiple segments and/or multiple indexes, each index potentially composed of a set of segments.

- Document Numbers
  - Internally, Lucene refers to documents by an integer document number.
  - The first document added to an index is numbered zero, and each subsequent document added gets a number one greater than the previous.

Index File Formats

- Each segment index maintains the following:
  - Segment info. This contains metadata about a segment, such as the number of documents, what files it uses, and information about how the segment is sorted
  - Field names. This contains metadata about the set of named fields used in the index.
  - Stored Field values. This contains, for each document, a list of attribute-value pairs, where the attributes are field names. These are used to store auxiliary information about the document, such as its title, url, or an identifier to access a database. The set of stored fields are what is returned for each hit when searching. This is keyed by document number.
  - Term dictionary. A dictionary containing all of the terms used in all of the indexed fields of all of the documents. The dictionary also contains the number of documents which contain the term, and pointers to the term's frequency and proximity data.
  - Term Frequency data. For each term in the dictionary, the numbers of all the documents that contain that term, and the frequency of the term in that document, unless frequencies are omitted (IndexOptions.DOCS)

Index File Formats

- Each segment index maintains the following:
  - Term Proximity data. For each term in the dictionary, the positions that the term occurs in each document. Note that this will not exist if all fields in all documents omit position data.
  - Normalization factors. For each field in each document, a value is stored that is multiplied into the score for hits on that field.
  - Term Vectors. For each field in each document, the term vector (sometimes called document vector) may be stored. A term vector consists of term text and term frequency. To add Term Vectors to your index see the Field constructors
  - Per-document values. Like stored values, these are also keyed by document number, but are generally intended to be loaded into main memory for fast access. Whereas stored values are generally intended for summary results from searches, per-document values are useful for things like scoring factors.
  - Live documents. An optional file indicating which documents are live.
  - Point values. Optional pair of files, recording dimensionally indexed fields, to enable fast numeric range filtering and large numeric values like BigInteger and BigDecimal (1D) and geographic shape intersection (2D, 3D).
  - Vector values. The vector format stores numeric vectors in a format optimized for random access and computation, supporting high-dimensional nearest-neighbor search.

Search and Scoring in Lucene

- Search Basics
  - Lucene offers a wide variety of Query implementations.

- To perform a search, applications usually call
  - `IndexSearcher.search(Query,int)`

- Once a Query has been created and submitted to the IndexSearcher, the scoring process begins.
  - After some infrastructure setup, control finally passes to the Weight implementation and its Scorer or BulkScorer instances.

Search and Scoring in Lucene

- Query Classes
- TermQuery
  - Of the various implementations of Query, the TermQuery is the easiest to understand and the most often used in applications.
  - A TermQuery matches all the documents that contain the specified Term, which is a word that occurs in a certain Field.
  - Thus, a TermQuery identifies and scores all Documents that have a Field with the specified string in it.
  - Constructing a TermQuery is as simple as:

  - `TermQuery tq = new TermQuery(new Term("fieldName", "term"));`

  - In this example, the Query identifies all Documents that have the Field named "fieldName" containing the word "term".

Search and Scoring in Lucene

- Query Classes
- BooleanQuery
  - Things start to get interesting when one combines multiple TermQuery instances into a BooleanQuery.
  - A BooleanQuery contains multiple BooleanClauses, where each clause contains a sub-query (Query instance) and an operator (from BooleanClause.Occur) describing how that sub-query is combined with the other clauses:
    - SHOULD — Use this operator when a clause can occur in the result set, but is not required. If a query is made up of all SHOULD clauses, then every document in the result set matches at least one of these clauses.
    - MUST — Use this operator when a clause is required to occur in the result set and should contribute to the score. Every document in the result set will match all such clauses.
    - FILTER — Use this operator when a clause is required to occur in the result set but should not contribute to the score. Every document in the result set will match all such clauses.
    - MUST NOT — Use this operator when a clause must not occur in the result set. No document in the result set will match any such clauses.

- Boolean queries are constructed by adding two or more BooleanClause instances.
  - If too many clauses are added, a TooManyClauses exception will be thrown during searching.
  - This most often occurs when a Query is rewritten into a BooleanQuery with many TermQuery clauses, for example by WildcardQuery.
  - The default setting for the maximum number of clauses is 1024, but this can be changed via the static method IndexSearcher.setMaxClauseCount(int).

Search and Scoring in Lucene

- Query Classes
- Phrases
  - Another common search is to find documents containing certain phrases.
  - This is handled in different ways:
    - PhraseQuery — Matches a sequence of Terms. PhraseQuery uses a slop factor to determine how many positions may occur between any two terms in the phrase and still be considered a match. The slop is 0 by default, meaning the phrase must match exactly.
    - MultiPhraseQuery — A more general form of PhraseQuery that accepts multiple Terms for a position in the phrase. For example, this can be used to perform phrase queries that also incorporate synonyms.
    - Interval queries in the Queries module.

Search and Scoring in Lucene

- Query Classes
- PointRangeQuery
  - The PointRangeQuery matches all documents that occur in a numeric range. For PointRangeQuery to work, you must index the values using a one of the numeric fields (IntPoint, LongPoint, FloatPoint, or DoublePoint).
- PrefixQuery, WildcardQuery, RegexpQuery
  - The PrefixQuery allows an application to identify all documents with terms that begin with a certain string.
  - The WildcardQuery generalizes this by allowing for the use of * (matches 0 or more characters) and ? (matches exactly one character) wildcards.
  - The RegexpQuery is even more general than WildcardQuery, allowing an application to identify all documents with terms that match a regular expression pattern.
- FuzzyQuery
  - A FuzzyQuery matches documents that contain terms similar to the specified term. Similarity is determined using Levenshtein distance. This type of query can be useful when accounting for spelling variations in the collection.

Scoring

- Lucene scoring is the heart of why we all love Lucene.

- Lucene scoring supports a number of pluggable information retrieval models, including:
  - Vector Space Model (VSM)
  - Probabilistic Models such as Okapi BM25 and DFR
  - Language models
- These models can be plugged in via the Similarity API, and offer extension hooks and parameters for tuning.

Scoring

- Scoring
  - Scoring is very much dependent on the way documents are indexed, so it is important to understand indexing.
  - Be sure to use the useful IndexSearcher.explain(Query, doc) to understand how the score for a certain matching document was computed.
  - Generally, the Query determines which documents match (a binary decision), while the Similarity determines how to assign scores to the matching documents.

- Fields and Documents
  - In Lucene, the objects we are scoring are Documents.
  - A Document is a collection of Fields.
  - It is important to note that Lucene scoring works on Fields and then combines the results to return Documents.
  - This is important because two Documents with the exact same content, but one having the content in two Fields and the other in one Field may return different scores for the same query due to length normalization.

- Score Boosting
  - Lucene allows influencing the score contribution of various parts of the query by wrapping with BoostQuery.

Changing Scoring - Similarity

- Changing the scoring formula
  - Changing Similarity is an easy way to influence scoring, this is done at index-time with IndexWriterConfig.setSimilarity(Similarity) and at query-time with IndexSearcher.setSimilarity(Similarity).
  - Be sure to use the same Similarity at query-time as at index-time (so that norms are encoded/decoded correctly); Lucene makes no effort to verify this.
  - You can influence scoring by configuring a different built-in Similarity implementation, or by tweaking its parameters, subclassing it to override behavior.

  - BM25Similarity is an optimized implementation of the successful Okapi BM25 model.
  - ClassicSimilarity is the original Lucene scoring function. It is based on the Vector Space Model. For more information, see TFIDFSimilarity.
  - SimilarityBase provides a basic implementation of the Similarity contract and exposes a highly simplified interface, which makes it an ideal starting point for new ranking functions.

Changing Scoring - Similarity

$$\sum_{t\ in\ q} tf\left(t\ in\ d\right)\cdot idf\left(t\right)\cdot boost\left(t.field\ in\ d\right)\cdot lengthNorm\left(t.field\ in\ d\right)$$

Table 3.5 Factors in the scoring formula

Factor	Description
`tf(t in d)`	Term frequency factor for the term (`t`) in the document (`d`).
`idf(t)`	Inverse document frequency of the term.
`boost(t.field in d)`	Field boost, as set during indexing.
`lengthNorm(t.field in d)`	Normalization value of a field, given the number of terms within the field. This value is computed during indexing and stored in the index.
`coord(q, d)`	Coordination factor, based on the number of query terms the document contains.
`queryNorm(q)`	Normalization value for a query, given the sum of the squared weights of each of the query terms.

Changing Scoring - Similarity

- Integrating field values into the score
  - The below query matches the same documents as originalQuery and computes scores as similarityScore + 0.7 * featureScore:

```
Query originalQuery = new BooleanQuery.Builder()
        .add(new TermQuery(new Term("body", "apache")), Occur.SHOULD)
        .add(new TermQuery(new Term("body", "lucene")), Occur.SHOULD)
        .build();
Query featureQuery = FeatureField.newSaturationQuery("features", "pagerank");
Query query = new BooleanQuery.Builder()
        .add(originalQuery, Occur.MUST)
        .add(new BoostQuery(featureQuery, 0.7f), Occur.SHOULD)
        .build();
```

Changing Scoring - Similarity

- Integrating field values into the score
  - The below example shows how to compute scores as similarityScore * Math.log(popularity) using the expressions module and assuming that values for the popularity field have been set in a NumericDocValuesField at index time:

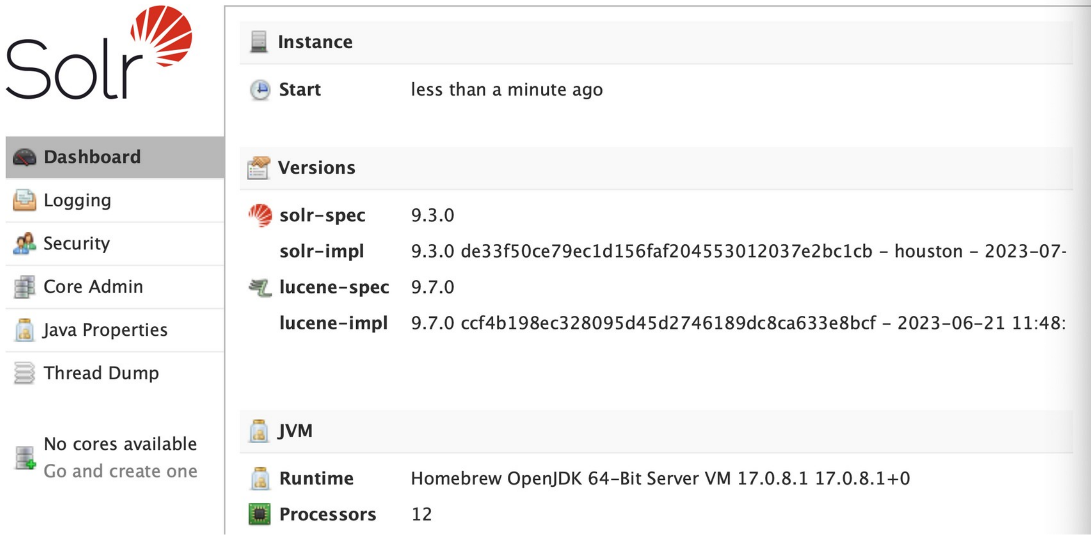
```
// compile an expression:
Expression expr = JavascriptCompiler.compile("_score * ln(popularity)");

// SimpleBindings just maps variables to SortField instances
SimpleBindings bindings = new SimpleBindings();
bindings.add(new SortField("_score", SortField.Type.SCORE));
bindings.add(new SortField("popularity", SortField.Type.INT));

// create a query that matches based on 'originalQuery' but
// scores using expr
Query query = new FunctionScoreQuery(
        originalQuery,
        expr.getDoubleValuesSource(bindings));
```

Apache Solr

- APACHE SOLR™
  - is the popular, blazing-fast, open source enterprise search platform built on Apache Lucene™.
  - is highly reliable, scalable and fault tolerant, providing distributed indexing, replication and load-balanced querying, automated failover and recovery, centralized configuration and more.
  - Solr powers the search and navigation features of many of the world's largest internet sites.

Solr

- Install Solr on Mac OS
  - `$ brew install solr`
- Launch Solr
  - `$ brew services start solr`
- Create a new core
  - `$ solr delete -c [my_solr]`
  - `$ solr create_core -c [y_sol`



Instance

Start	less than a minute ago

Versions

solr-spec	9.3.0
solr-impl	9.3.0 de33f50ce79ec1d156faf204553012037e2bc1cb - houston - 2023-07-
lucene-spec	9.7.0
lucene-impl	9.7.0 ccf4b198ec328095d45d2746189dc8ca633e8bcf - 2023-06-21 11:48:

JVM

Runtime	Homebrew OpenJDK 64-Bit Server VM 17.0.8.1 17.0.8.1+0
Processors	12

Using SolrJ

- SolrJ
  - is an API that makes it easy for applications written in Java (or any language based on the JVM) to talk to Solr.

- For projects built with Maven, place the following in your `pom.xml`:

```
<dependency>
  <groupId>org.apache.solr</groupId>
  <artifactId>solr-solrj</artifactId>
  <version>9.3.0</version>
</dependency>
```

Using SolrJ

- Querying.java

```
public class Querying {
    public static void main(String[] args) throws IOException, SolrServerException {
        final SolrClient client = getSolrClient();

        final Map<String, String> queryParamMap = new HashMap<String, String>();
        queryParamMap.put("q", "*:*");
        queryParamMap.put("fl", "id, name");
        queryParamMap.put("sort", "id asc");
        MapSolrParams queryParams = new MapSolrParams(queryParamMap);

        final QueryResponse response = client.query("techproducts", queryParams);
        final SolrDocumentList documents = response.getResults();

        System.out.println("Found " + documents.getNumFound() + " documents");
        for (SolrDocument document : documents) {
            final String id = (String) document.getFirstValue("id");
            final String name = (String) document.getFirstValue("name");

            System.out.println("id: " + id + "; name: " + name);
        }
    }
```

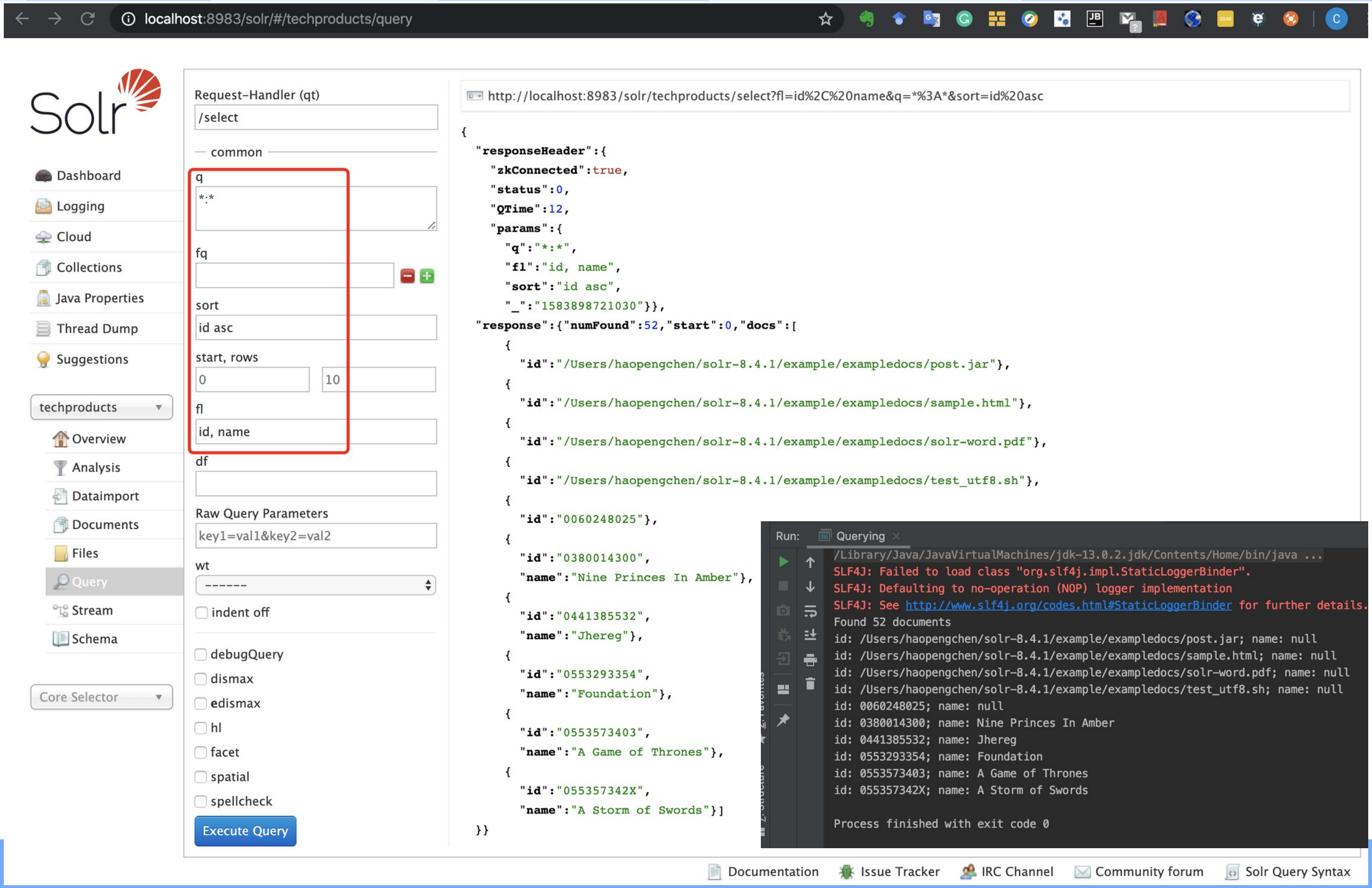
Using SolrJ

- Querying.java

```
    public static SolrClient getSolrClient() {
        final String solrUrl = "http://localhost:8983/solr";
        return new HttpSolrClient.Builder(solrUrl)
                .withConnectionTimeout(10000)
                .withSocketTimeout(60000)
                .build();
    }

}
```

Using SolrJ

REliable, INtelligent & Scalable Systems



Request-Handler (qt): /select

common

q: *:*

fq:

sort: id asc

start, rows: 0 10

fl: id, name

df:

Raw Query Parameters: key1=val1&key2=val2

wt: ------

☐ indent off

☐ debugQuery

☐ dismax

☐ edismax

☐ hl

☐ facet

☐ spatial

☐ spellcheck

http://localhost:8983/solr/techproducts/select?fl=id%2C%20name&q=*%3A*&sort=id%20asc

```
{
  "responseHeader":{
    "zkConnected":true,
    "status":0,
    "QTime":12,
    "params":{
      "q":"*:*",
      "fl":"id, name",
      "sort":"id asc",
      "_":"1583898721030"}},
  "response":{"numFound":52,"start":0,"docs":[
      {
        "id":"/Users/haopengchen/solr-8.4.1/example/exampledocs/post.jar"},
      {
        "id":"/Users/haopengchen/solr-8.4.1/example/exampledocs/sample.html"},
      {
        "id":"/Users/haopengchen/solr-8.4.1/example/exampledocs/solr-word.pdf"},
      {
        "id":"/Users/haopengchen/solr-8.4.1/example/exampledocs/test_utf8.sh"},
      {
        "id":"0060248025"},
      {
        "id":"0380014300",
        "name":"Nine Princes In Amber"},
      {
        "id":"0441385532",
        "name":"Jhereg"},
      {
        "id":"0553293354",
        "name":"Foundation"},
      {
        "id":"0553573403",
        "name":"A Game of Thrones"},
      {
        "id":"055357342X",
        "name":"A Storm of Swords"}]
  }}
```

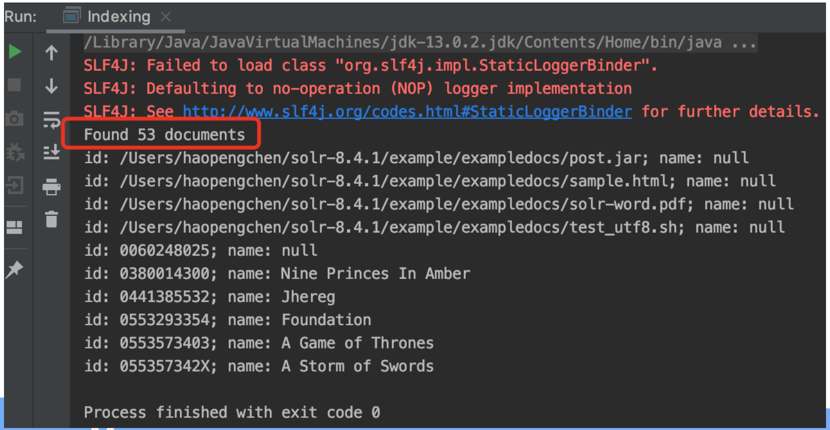
Run: Querying

```
/Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home/bin/java ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Found 52 documents
id: /Users/haopengchen/solr-8.4.1/example/exampledocs/post.jar; name: null
id: /Users/haopengchen/solr-8.4.1/example/exampledocs/sample.html; name: null
id: /Users/haopengchen/solr-8.4.1/example/exampledocs/solr-word.pdf; name: null
id: /Users/haopengchen/solr-8.4.1/example/exampledocs/test_utf8.sh; name: null
id: 0060248025; name: null
id: 0380014300; name: Nine Princes In Amber
id: 0441385532; name: Jhereg
id: 0553293354; name: Foundation
id: 0553573403; name: A Game of Thrones
id: 055357342X; name: A Storm of Swords

Process finished with exit code 0
```

Indexing

- Indexing.java

```
final SolrInputDocument doc = new SolrInputDocument();
doc.addField(“id”, UUID.randomUUID().toString());
doc.addField(“name”, “Amazon Kindle Paperwhite”);

final UpdateResponse updateResponse = client.add("techproducts", doc);
// Indexed documents must be committed
client.commit("techproducts");
```



```
/Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home/bin/java ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Found 53 documents
id: /Users/haopengchen/solr-8.4.1/example/exampledocs/post.jar; name: null
id: /Users/haopengchen/solr-8.4.1/example/exampledocs/sample.html; name: null
id: /Users/haopengchen/solr-8.4.1/example/exampledocs/solr-word.pdf; name: null
id: /Users/haopengchen/solr-8.4.1/example/exampledocs/test_utf8.sh; name: null
id: 0060248025; name: null
id: 0380014300; name: Nine Princes In Amber
id: 0441385532; name: Jhereg
id: 0553293354; name: Foundation
id: 0553573403; name: A Game of Thrones
id: 055357342X; name: A Storm of Swords

Process finished with exit code 0
```

Java Object Binding

REliable, INtelligent & Scalable Systems

- TechProduct.java

```
public class TechProduct {
    @Field
    public String id;
    @Field
    public String name;

    public TechProduct(String id, String name) {
        this.id = id;
        this.name = name;
    }

    public TechProduct() { }

    public String getId() {
        return this.id;
    }

    public String getName() {
        return this.name;
    }
}
```

Java Object Binding

RElíable, INtelligent & Scalable Systems

- ObjectBinding.java

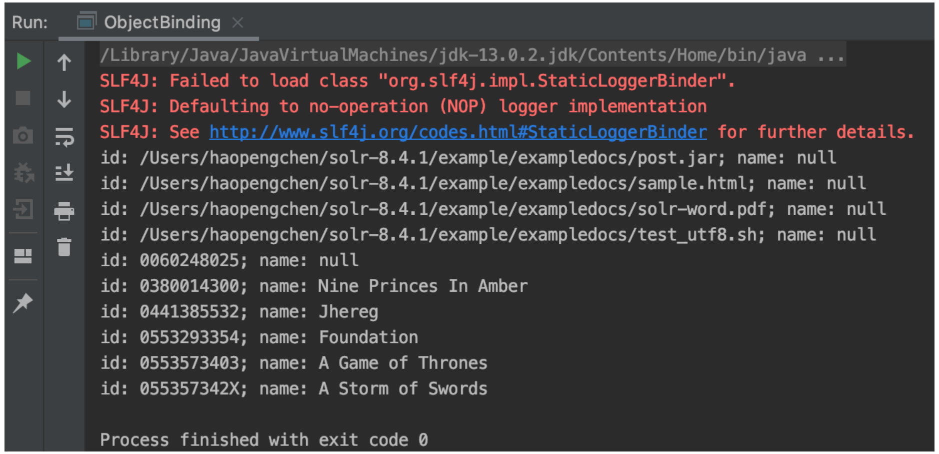
```
public class ObjectBinding {
    public static void main(String[] args) throws IOException, SolrServerException {
        final SolrClient client = getSolrClient();
        final TechProduct kindle = new TechProduct("kindle-id-4", "Amazon Kindle 
Paperwhite");
        final UpdateResponse response = client.addBean("techproducts", kindle);
        client.commit("techproducts");

        final SolrQuery query = new SolrQuery("*:*");
        query.addField("id");
        query.addField("name");
        query.setSort("id", SolrQuery.ORDER.asc);

        final QueryResponse responseOne = client.query("techproducts", query);
        final List<TechProduct> products = responseOne.getBeans(TechProduct.class);
        for (TechProduct product : products) {
            final String id = product.getId();
            final String name = product.getName();
            System.out.println("id: " + id + "; name: " + name);
        }
    }
```

Java Object Binding

REliable, INtelligent & Scalable Systems

- ObjectBinding.java

```
public static SolrClient getSolrClient() {
    final String solrUrl = "http://localhost:8983/solr";
    return new HttpSolrClient.Builder(solrUrl)
            .withConnectionTimeout(10000)
            .withSocketTimeout(60000)
            .build();
}
}
```



```
/Library/Java/JavaVirtualMachines/jdk-13.0.2.jdk/Contents/Home/bin/java ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
id: /Users/haopengchen/solr-8.4.1/example/exampledocs/post.jar; name: null
id: /Users/haopengchen/solr-8.4.1/example/exampledocs/sample.html; name: null
id: /Users/haopengchen/solr-8.4.1/example/exampledocs/solr-word.pdf; name: null
id: /Users/haopengchen/solr-8.4.1/example/exampledocs/test_utf8.sh; name: null
id: 0060248025; name: null
id: 0380014300; name: Nine Princes In Amber
id: 0441385532; name: Jhereg
id: 0553293354; name: Foundation
id: 0553573403; name: A Game of Thrones
id: 055357342X; name: A Storm of Swords

Process finished with exit code 0
```

Elasticsearch

- Elasticsearch is a highly scalable open-source full-text search and analytics engine.
  - It allows you to store, search, and analyze big volumes of data quickly and in near real time.
  - It is generally used as the underlying engine/technology that powers applications that have complex search features and requirements.

  - https://www.elastic.co

Elasticsearch – Basic Concepts

- Near Real Time
  - Elasticsearch is a near real time search platform. What this means is there is a slight latency (normally one second) from the time you index a document until the time it becomes searchable.
- Document
  - A document is a basic unit of information that can be indexed.
  - This document is expressed in JSON (JavaScript Object Notation) which is a ubiquitous internet data interchange format.
- Index
  - An index is a collection of documents that have somewhat similar characteristics.
  - An index is identified by a name (that must be all lowercase) and this name is used to refer to the index when performing indexing, search, update, and delete operations against the documents in it.

References

- Apache Lucene
  - http://lucene.apache.org/
- Lucene in Action
  - By Otis Gospodnetic & Erik Hatcher
  - MANNING Publishing
- Lucene 9.7.0 demo API
  - https://lucene.apache.org/core/9_7_0/demo/index.html
- Apache Solr
  - https://lucene.apache.org/solr/
- SolrJ
  - https://solr.apache.org/guide/solr/latest/deployment-guide/solrj.html

Thank You!