newthinking
store

cloudera

O’REILLY®
www.oreilly.de

Lucene Java 2.9:
Numeric Search, Per-Segment Search, Near-Real-Time Search, and the new TokenStream API

Uwe Schindler

Lucene Java Committer

uschindler@apache.org

PANGAEA® - Publishing Network for Geoscientific & Environmental Data (www.pangaea.de)
MARUM, Center for Marine Environmental Sciences, Bremen, Germany (www.marum.de)

New features in Lucene Java 2.9

- **Lucene now includes high-performance handling of numeric fields. Such fields are indexed with a trie structure, enabling simple to use and much faster numeric range searching without having to externally pre-process numeric values into textual values**
- **Smarter, more scalable multi-term queries (wildcard, range, etc)**
- **Per segment searching and caching (can lead to much faster reopen among other things)**
- **A freshly optimized Collector/Scorer API**
- **Scoring is now optional when sorting by field, or using a custom Collector, gaining sizable performance when scores are not required**
- **Near real-time search capabilities added to IndexWriter**
- **A new Attribute based TokenStream API**
- **A new QueryParser framework in contrib with a core QueryParser replacement impl included**
- New Query types
- Improved Unicode support and the addition of Collation contrib
- New analyzers (PersianAnalyzer, ArabicAnalyzer, SmartChineseAnalyzer)
- New fast-vector-highlighter

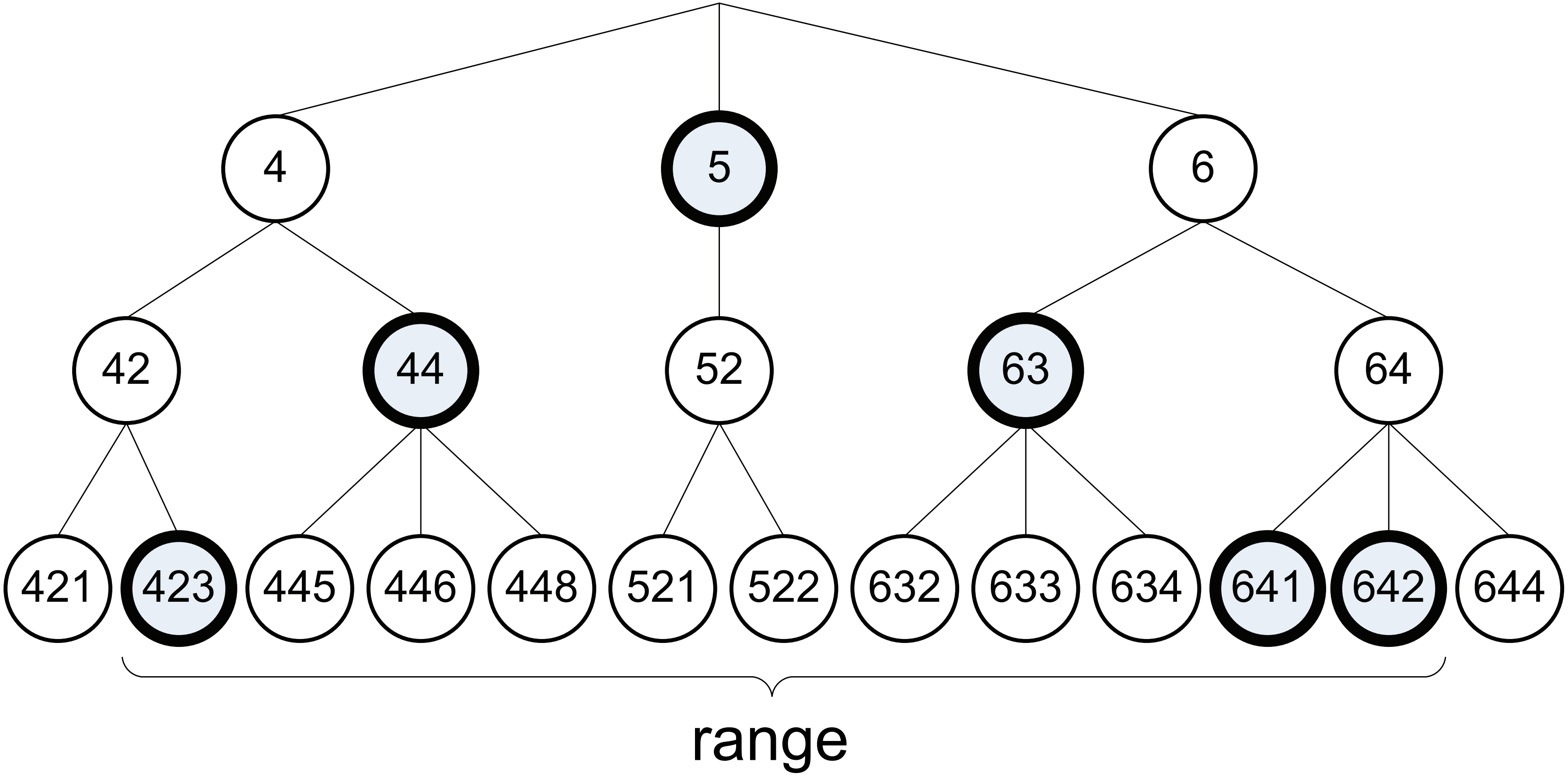
Source: Release notes of Lucene Java 2.9

Numeric Fields

Problems with 2.4’s RangeQueries/-Filters

- Classical **RangeQuery** hits TooManyClausesException on large ranges and is very slow.
- **ConstantScoreRangeQuery** is faster, cacheable, but still has to visit a large number of terms.
- Both need to enumerate **a large number of terms** from **TermEnum** and then retrieve **TermDocs** for each term.
- The number of terms to visit grows with number of documents and unique values in index (especially for float/double values)

TrieRange: How it works



Supported Data Types

- *Native data type:* `long`, `int` (standard Java signed). All “tricks” like padding are **not needed!** These types are internally made unsigned, each trie precision is generated by stripping off least significant bits (using `precisionStep` parameter). Each value is then converted to a sequence of 7bit ASCII chars, result is prefixed with the number of bits stripped, and indexed as term. Only 7 bits/char are used because of most efficient bit layout in index (8 or more bits would split into two or more bytes when UTF-8 encoded).
- `double`, `float`: Converter to/from IEEE-754 bit layout that sorts like a signed `long`/`int`
- `Date`/`Calendar`: Convert to UNIX time stamp with e.g. `Date.getTime()`

Speed

- Upper limit on number of terms, independent of index size. This value depends only on `precisionStep`
- *Term numbers:* 8bit approx. 400 terms, 4 bit approx. 100 terms, 2 bit approx. 40 terms
- *Query time:* in most cases <100 ms with 1,000,000 docs index, 13 numeric fields, `precisionStep` 8 bit

How to use (indexing)

- New convenience class **NumericField** that optionally also stores the numeric value as string. Provides various setters for different data types.
- The work is done by **NumericTokenStream**, which “tokenizes” the number into the binary encoded trie terms.

```
Directory directory = new RAMDirectory();
Analyzer analyzer = new WhitespaceAnalyzer();
IndexWriter writer = new IndexWriter(directory, analyzer,
     IndexWriter.MaxFieldLength.UNLIMITED);
for (int i = 0; i < 20000; i++) {
     Document doc = new Document();
     doc.add(new Field("id", String.valueOf(i),
          Field.Store.YES, Field.Index.NOT_ANALYZED_NO_NORMS));
     doc.add(new NumericField("newNumeric", 4,
          Field.Store.YES, true).setIntValue(i));
     writer.addDocument(doc);
}
writer.close();
```

How to use (searching)

- New classes: **NumericRangeQuery**, **NumericRangeFilter** with “static” ctors per data type.
- Old RangeQuery & co. is deprecated and replaced by **TermRangeQuery**
- 4 different modes for **MultiTermQueries**: Conventional with scoring (not recommended), constant score with Filter or BooleanQuery, automatic constant score dependent on term count. WildcardQuery, PrefixQuery and FuzzyQuery are MTQs since 2.9, too.

```
IndexSearcher searcher = new IndexSearcher(directory, true);
Query query = NumericRangeQuery.newIntRange(
     "newNumeric", 4, 10, 10000, true, false);
TopDocs docs = searcher.search(query, null, 10);
assertNotNull("Docs is null", docs);
assertEquals(9990, docs.totalHits);
for (int i = 0; i < docs.scoreDocs.length; i++) {
     ScoreDocs d= docs.scoreDocs[i];
     assertTrue(sd.doc >= 10 && sd.doc < 10000);
}
```

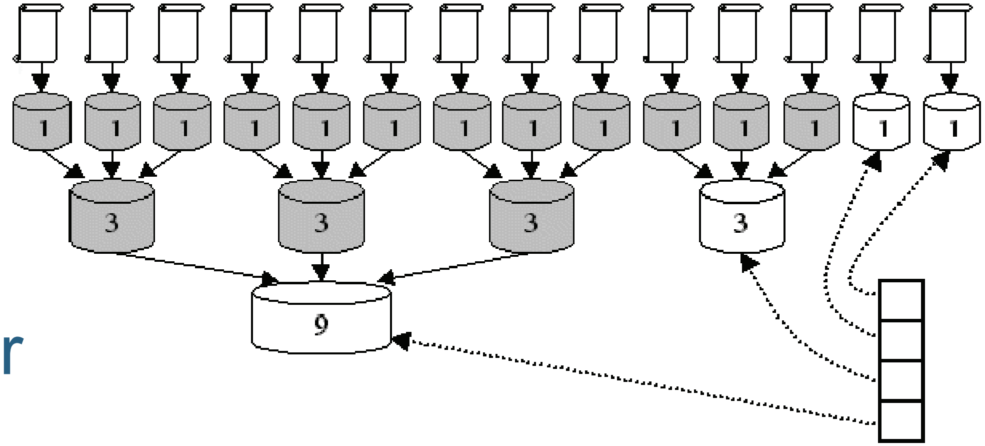
NumericField and FieldCache

- **NumericFields** can be loaded into **FieldCache** and will be used for sorting.
- **FieldCache.AUTO** / **SortField.AUTO** deprecated.
- New range filter implementation based completely on using the FieldCache: **FieldCacheRangeFilter**. Similar API like NumericRangeFilter, but also supports string(index) fields.

Per-Segment Search

Segments in Lucene

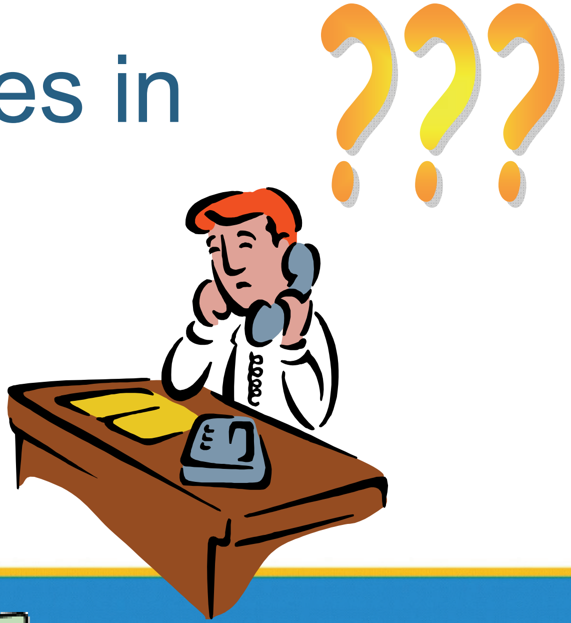
- Each index consists of various segments placed in the index directory. All documents are added to new in-RAM segment files, merged to on-disk files after flushing *(each document is initially one segment!)*.



- Lucene writes segments incrementally and then can merge them.
- Optimized index consists of one segment.
- **IndexReader.reopen()** adds new/changed segments after commit to segments of an already existing IndexReader (lower I/O cost in contrast to re-opening the whole IndexReader).

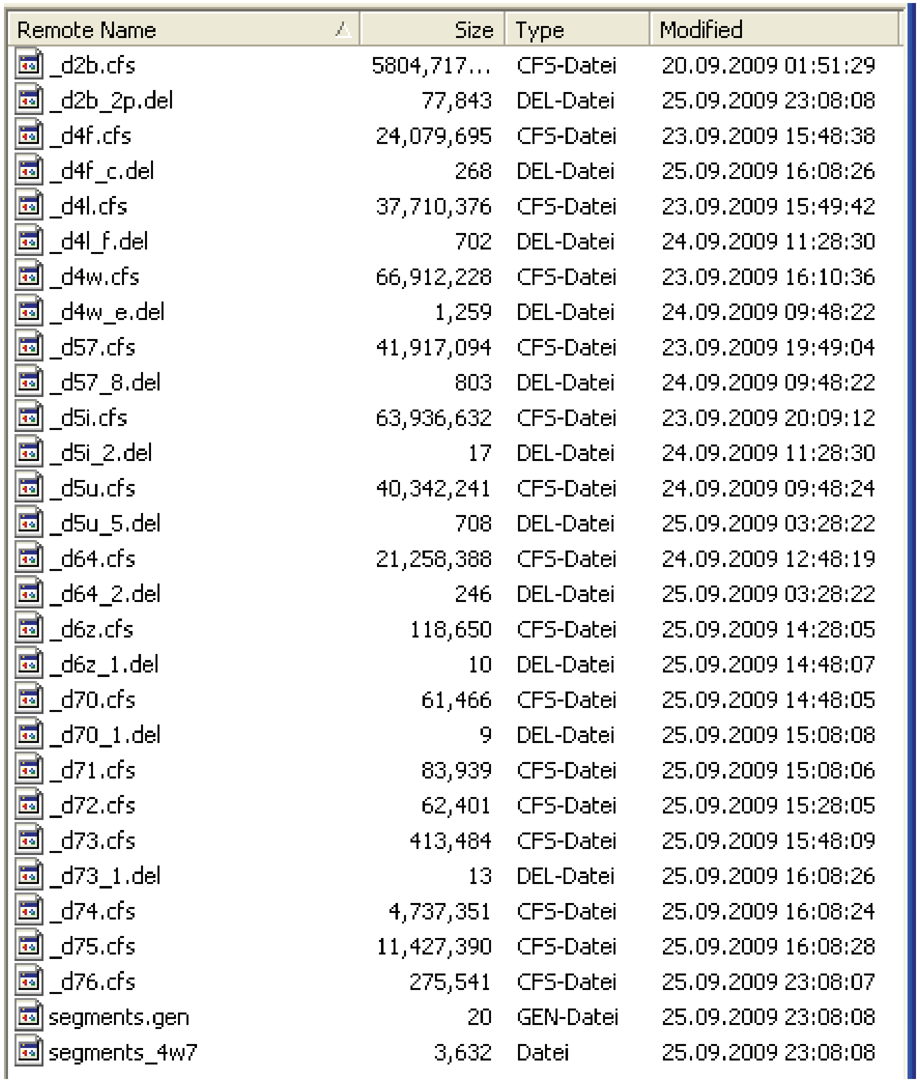
Problems

- **FieldCache** used for sorting is keyed against the IndexReader instance.
- After reopen the whole FieldCache is invalid and needs to be reloaded.
- Long “warming” time for sorted queries (and also function queries in Solr) after reopen.



What has changed?

- **IndexSearcher** now works directly on segments (cf. MultiSearcher), results are merged by **Collectors** (**TopDocsCollector**,…). Non-expert API stays unchanged.
- **FieldCache** therefore also works on segments ⇨ sorting warmup after reopening IndexReaders is much faster, as only FieldCaches for new/changed segments have to be rebuilt.
- Scoring decoupled from **Collector.**



Remote Name	Size	Type	Modified
_d2b.cfs	5804,717...	CFS-Datei	20.09.2009 01:51:29
_d2b_2p.del	77,843	DEL-Datei	25.09.2009 23:08:08
_d4f.cfs	24,079,695	CFS-Datei	23.09.2009 15:48:38
_d4f_c.del	268	DEL-Datei	25.09.2009 16:08:26
_d4l.cfs	37,710,376	CFS-Datei	23.09.2009 15:49:42
_d4l_f.del	702	DEL-Datei	24.09.2009 11:28:30
_d4w.cfs	66,912,228	CFS-Datei	23.09.2009 16:10:36
_d4w_e.del	1,259	DEL-Datei	24.09.2009 09:48:22
_d57.cfs	41,917,094	CFS-Datei	23.09.2009 19:49:04
_d57_8.del	803	DEL-Datei	24.09.2009 09:48:22
_d5i.cfs	63,936,632	CFS-Datei	23.09.2009 20:09:12
_d5i_2.del	17	DEL-Datei	24.09.2009 11:28:30
_d5u.cfs	40,342,241	CFS-Datei	24.09.2009 09:48:24
_d5u_5.del	708	DEL-Datei	25.09.2009 03:28:22
_d64.cfs	21,258,388	CFS-Datei	24.09.2009 12:48:19
_d64_2.del	246	DEL-Datei	25.09.2009 03:28:22
_d6z.cfs	118,650	CFS-Datei	25.09.2009 14:28:05
_d6z_1.del	10	DEL-Datei	25.09.2009 14:48:07
_d70.cfs	61,466	CFS-Datei	25.09.2009 14:48:05
_d70_1.del	9	DEL-Datei	25.09.2009 15:08:08
_d71.cfs	83,939	CFS-Datei	25.09.2009 15:08:06
_d72.cfs	62,401	CFS-Datei	25.09.2009 15:28:05
_d73.cfs	413,484	CFS-Datei	25.09.2009 15:48:09
_d73_1.del	13	DEL-Datei	25.09.2009 16:08:26
_d74.cfs	4,737,351	CFS-Datei	25.09.2009 16:08:24
_d75.cfs	11,427,390	CFS-Datei	25.09.2009 16:08:28
_d76.cfs	275,541	CFS-Datei	25.09.2009 23:08:07
segments.gen	20	GEN-Datei	25.09.2009 23:08:08
segments_4w7	3,632	Datei	25.09.2009 23:08:08

New **Collector** class

- Replacement for **HitCollector**.
- Gets notification about **IndexReader** change *(together with new document ID base).* This method can be used to change FieldCache arrays used during collecting to new IR.
- *collect()* method gets document ID from current reader. It may map it by adding the current base to get a global ID.
- *collect()* method no longer gets a score. It gets a notification about change of the underlying Scorer instance and can call **Scorer**.*score()* if needed. This can be used to skip scoring for queries that don't care about score.
- Old **HitCollectors** can be wrapped by a special **HitCollectorWrapper** *(they get called with rebased doc IDs and wrapper calls Scorer.score() for each hit).*

Near-Real-Time Search

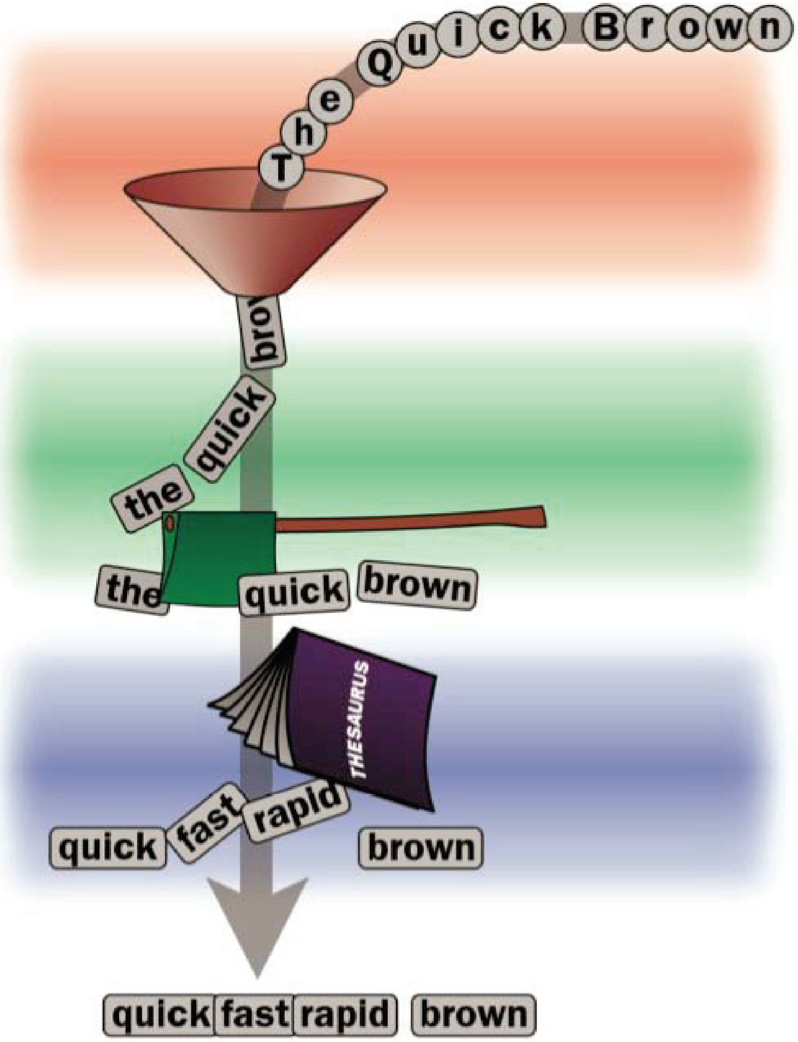
NRT additions

- Directly get an **IndexReader** from **IndexWriter** containing also uncommitted (in-memory) changes: **IndexWriter**.*getReader()*
- Supports *reopen()* as usual.
- Callback for warming merged segments: **IndexWriter**.*setMergedSegmentWarmer()*

New Attribute-based TokenStream API

Tokenizers, TokenFilters, TokenStreams

- **TokenStream** is base class for **Tokenizer** and **TokenFilter**
- Decorator pattern *(TokenFilter adds functionality to a Tokenizer)*
- Implementation part of each **Tokenizer** / **TokenFilter** should be final
- *Lucene 2.4:* **Token** class holds all **attributes** of a token: term, position increment, start/end offset, type and flags *(e.g. part of speech information passed between TokenFilters)*, payload



Lucene 2.9: **Attributes** instead of **Tokens**

- Introduces stronger typing and arbitrary attributes into the analysis process
- Easier to code custom **TokenStreams** by focusing only on needed attributes
- Helps set Lucene up for more flexible indexing options in the near future (LUCENE-1458)
- *Downside:* Some extra work transitioning your existing **TokenStreams** for 3.0:

  next(Token) ⇨ *incrementToken()*

Rewrite old ⇨ new TokenFilter

```
public final class LengthFilter extends TokenFilter {

  private final int min;
  private final int max;

  public LengthFilter(TokenStream in, int min, int max) {
    super(in);
    this.min = min;
    this.max = max;
  }

  public Token next(final Token reusableToken) throws IOException {
    for (Token nextToken = input.next(reusableToken);
    nextToken != null; nextToken = input.next(reusableToken)) {
      int len = nextToken.termLength();
      if (len >= min && len <= max) {
          return nextToken;
      }
    }
    return null;
  }
}
```

Rewrite old ⇨ new TokenFilter

```
public final class LengthFilter extends TokenFilter {

  private final int min;
  private final int m

  public LengthFilte
    super(in);
    this.min = min;
    this.max = max;
  }

  public Token next(
    for (Token nextT
    nextToken != nul
      int len = next
      if (len >= min
          return nex
      }
    }
    return null;
  }
}
```

```
public final class LengthFilter extends TokenFilter {

  private final int min;
  private final int max;

  private final TermAttribute termAtt;

  public LengthFilter(TokenStream in, int min, int max) {
    super(in);
    this.min = min;
    this.max = max;
    termAtt = (TermAttribute) addAttribute(TermAttribute.class);
  }

  public boolean incrementToken() throws IOException {
    while (input.incrementToken()) {
      int len = termAtt.termLength();
      if (len >= min && len <= max) {
          return true;
      }
    }
    return false;
  }
}
```

Rewrite old ⇨ new TokenFilter

```
public final class LengthFilter extends TokenFilter {

  private final int min;
  private final int r

  public LengthFilte
    super(in);
    this.min = min;
    this.max = max;
  }

  public Token next(
    for (Token nextT
    nextToken != nul
      int len = next
      if (len >= min
          return nex
      }
    }
    return null;
  }
}
```

```
public final class LengthFilter extends TokenFilter {

  private final int min;
  private final int max;

  private final TermAttribute termAtt;

  public LengthFilter(TokenStream in, int min, int max) {
    super(in);
    this.min = min;
    this.max = max;
    termAtt =                addAttribute(TermAttribute.class);
  }

  public boolean incrementToken() throws IOException {
    while (input.incrementToken()) {
      int len = termAtt.termLength();
      if (len >= min && len <= max) {
          return true;
      }
    }
    return false;
  }
}
```

Lucene 3.0: Generics

Websites

- NumericRangeQuery example: www.pangaea.de

  (PANGAEA® - Publishing Network for Geoscientific & Environmental Data)

- PANGAEA Framework for Metadata Portals: www.panFMP.org

- Lucene Java 2.9.0: lucene.apache.org/java/docs/

Happy coding with **Lucene 2.9.0**!

Thank You!