Branch: master ▾ Find file Copy path

lucene-solr / lucene / highlighter / src / java / org / apache / lucene / search / uhighlight / UnifiedHighlighter.java

romseygeek LUCENE-9062: QueryVisitor.consumeTermsMatching (#1037)

bed694e on Nov 28, 2019

4 contributors

Raw Blame History

1150 lines (1025 sloc) 46.6 KB

```
/*
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package org.apache.lucene.search.uhighlight;

import java.io.IOException;
import java.text.BreakIterator;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.EnumSet;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Locale;
import java.util.Map;
import java.util.Objects;
import java.util.Set;
import java.util.SortedSet;
import java.util.TreeSet;
import java.util.function.Predicate;
import java.util.function.Supplier;
```

```
36
37  import org.apache.lucene.analysis.Analyzer;
38  import org.apache.lucene.document.FieldType;
39  import org.apache.lucene.index.BaseCompositeReader;
40  import org.apache.lucene.index.FieldInfo;
41  import org.apache.lucene.index.FieldInfos;
42  import org.apache.lucene.index.Fields;
43  import org.apache.lucene.index.FilterLeafReader;
44  import org.apache.lucene.index.IndexOptions;
45  import org.apache.lucene.index.IndexReader;
46  import org.apache.lucene.index.LeafReader;
47  import org.apache.lucene.index.LeafReaderContext;
48  import org.apache.lucene.index.MultiReader;
49  import org.apache.lucene.index.ReaderUtil;
50  import org.apache.lucene.index.StoredFieldVisitor;
51  import org.apache.lucene.index.Term;
52  import org.apache.lucene.search.DocIdSetIterator;
53  import org.apache.lucene.search.IndexSearcher;
54  import org.apache.lucene.search.MatchAllDocsQuery;
55  import org.apache.lucene.search.MatchNoDocsQuery;
56  import org.apache.lucene.search.MultiTermQuery;
57  import org.apache.lucene.search.Query;
58  import org.apache.lucene.search.QueryVisitor;
59  import org.apache.lucene.search.ScoreDoc;
60  import org.apache.lucene.search.TopDocs;
61  import org.apache.lucene.search.Weight;
62  import org.apache.lucene.search.spans.SpanQuery;
63  import org.apache.lucene.util.BytesRef;
64  import org.apache.lucene.util.InPlaceMergeSorter;
65
66  /**
67   * A Highlighter that can get offsets from either
68   * postings ({@link IndexOptions#DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS}),
69   * term vectors ({@link FieldType#setStoreTermVectorOffsets(boolean)}),
70   * or via re-analyzing text.
71   * <p>
72   * This highlighter treats the single original document as the whole corpus, and then scores ir
73   * passages as if they were documents in this corpus. It uses a {@link BreakIterator} to find
74   * passages in the text; by default it breaks using {@link BreakIterator#getSentenceInstance(Lo
75   * getSentenceInstance(Locale.ROOT)}. It then iterates in parallel (merge sorting by offset) th
76   * the positions of all terms from the query, coalescing those hits that occur in a single pass
77   * into a {@link Passage}, and then scores each Passage using a separate {@link PassageScorer}
78   * Passages are finally formatted into highlighted snippets with a {@link PassageFormatter}.
79   * <p>
80   * You can customize the behavior by calling some of the setters, or by subclassing and overrid
81   * Some important hooks:
82   * <ul>
83   * <li>{@link #getBreakIterator(String)}: Customize how the text is divided into passages.
84   * <li>{@link #getScorer(String)}: Customize how passages are ranked.
85   * <li>{@link #getFormatter(String)}: Customize how snippets are formatted.
86   * </ul>
87   * <p>
```

```
 * This is thread-safe.
 *
 * @lucene.experimental
 */
public class UnifiedHighlighter {

  protected static final char MULTIVAL_SEP_CHAR = (char) 0;

  public static final int DEFAULT_MAX_LENGTH = 10000;

  public static final int DEFAULT_CACHE_CHARS_THRESHOLD = 524288; // ~ 1 MB (2 byte chars)

  static final IndexSearcher EMPTY_INDEXSEARCHER;

  static {
    try {
      IndexReader emptyReader = new MultiReader();
      EMPTY_INDEXSEARCHER = new IndexSearcher(emptyReader);
      EMPTY_INDEXSEARCHER.setQueryCache(null);
    } catch (IOException bogus) {
      throw new RuntimeException(bogus);
    }
  }

  protected static final LabelledCharArrayMatcher[] ZERO_LEN_AUTOMATA_ARRAY = new LabelledCharA

  protected final IndexSearcher searcher; // if null, can only use highlightWithoutSearcher

  protected final Analyzer indexAnalyzer;

  private boolean defaultHandleMtq = true; // e.g. wildcards

  private boolean defaultHighlightPhrasesStrictly = true; // AKA "accuracy" or "query debuggin

  private boolean defaultPassageRelevancyOverSpeed = true; //For analysis, prefer MemoryIndexO

  private int maxLength = DEFAULT_MAX_LENGTH;

  // BreakIterator is stateful so we use a Supplier factory method
  private Supplier<BreakIterator> defaultBreakIterator = () -> BreakIterator.getSentenceInstan

  private Predicate<String> defaultFieldMatcher;

  private PassageScorer defaultScorer = new PassageScorer();

  private PassageFormatter defaultFormatter = new DefaultPassageFormatter();

  private int defaultMaxNoHighlightPassages = -1;

  // lazy initialized with double-check locking; protected so subclass can init
  protected volatile FieldInfos fieldInfos;

```

```
140   private int cacheFieldValCharsThreshold = DEFAULT_CACHE_CHARS_THRESHOLD;
141
142   /**
143    * Extracts matching terms after rewriting against an empty index
144    */
145   protected static Set<Term> extractTerms(Query query) throws IOException {
146     Set<Term> queryTerms = new HashSet<>();
147     EMPTY_INDEXSEARCHER.rewrite(query).visit(QueryVisitor.termCollector(queryTerms));
148     return queryTerms;
149   }
150
151   /**
152    * Constructs the highlighter with the given index searcher and analyzer.
153    *
154    * @param indexSearcher Usually required, unless {@link #highlightWithoutSearcher(String, Que
155    *                      used, in which case this needs to be null.
156    * @param indexAnalyzer Required, even if in some circumstances it isn't used.
157    */
158   public UnifiedHighlighter(IndexSearcher indexSearcher, Analyzer indexAnalyzer) {
159     this.searcher = indexSearcher; //TODO: make non nullable
160     this.indexAnalyzer = Objects.requireNonNull(indexAnalyzer,
161         "indexAnalyzer is required"
162             + " (even if in some circumstances it isn't used)");
163   }
164
165   public void setHandleMultiTermQuery(boolean handleMtq) {
166     this.defaultHandleMtq = handleMtq;
167   }
168
169   public void setHighlightPhrasesStrictly(boolean highlightPhrasesStrictly) {
170     this.defaultHighlightPhrasesStrictly = highlightPhrasesStrictly;
171   }
172
173   public void setMaxLength(int maxLength) {
174     if (maxLength < 0 || maxLength == Integer.MAX_VALUE) {
175       // two reasons: no overflow problems in BreakIterator.preceding(offset+1),
176       // our sentinel in the offsets queue uses this value to terminate.
177       throw new IllegalArgumentException("maxLength must be < Integer.MAX_VALUE");
178     }
179     this.maxLength = maxLength;
180   }
181
182   public void setBreakIterator(Supplier<BreakIterator> breakIterator) {
183     this.defaultBreakIterator = breakIterator;
184   }
185
186   public void setScorer(PassageScorer scorer) {
187     this.defaultScorer = scorer;
188   }
189
190   public void setFormatter(PassageFormatter formatter) {
191     this.defaultFormatter = formatter;
```

```
192   }
193
194   public void setMaxNoHighlightPassages(int defaultMaxNoHighlightPassages) {
195     this.defaultMaxNoHighlightPassages = defaultMaxNoHighlightPassages;
196   }
197
198   public void setCacheFieldValCharsThreshold(int cacheFieldValCharsThreshold) {
199     this.cacheFieldValCharsThreshold = cacheFieldValCharsThreshold;
200   }
201
202   public void setFieldMatcher(Predicate<String> predicate) {
203     this.defaultFieldMatcher = predicate;
204   }
205
206   /**
207    * Returns whether {@link MultiTermQuery} derivatives will be highlighted.  By default it's e
208    * highlighting can be expensive, particularly when using offsets in postings.
209    */
210   protected boolean shouldHandleMultiTermQuery(String field) {
211     return defaultHandleMtq;
212   }
213
214   /**
215    * Returns whether position sensitive queries (e.g. phrases and {@link SpanQuery}ies)
216    * should be highlighted strictly based on query matches (slower)
217    * versus any/all occurrences of the underlying terms.  By default it's enabled, but there's
218    * queries aren't used.
219    */
220   protected boolean shouldHighlightPhrasesStrictly(String field) {
221     return defaultHighlightPhrasesStrictly;
222   }
223
224
225   protected boolean shouldPreferPassageRelevancyOverSpeed(String field) {
226     return defaultPassageRelevancyOverSpeed;
227   }
228
229   /**
230    * Returns the predicate to use for extracting the query part that must be highlighted.
231    * By default only queries that target the current field are kept. (AKA requireFieldMatch)
232    */
233   protected Predicate<String> getFieldMatcher(String field) {
234     if (defaultFieldMatcher != null) {
235       return defaultFieldMatcher;
236     } else {
237       // requireFieldMatch = true
238       return (qf) -> field.equals(qf);
239     }
240   }
241
242   /**
243    * The maximum content size to process.  Content will be truncated to this size before highli
```

```
244     * snippets closer to the beginning of the document better summarize its content.
245     */
246    public int getMaxLength() {
247      return maxLength;
248    }
249
250    /**
251     * Returns the {@link BreakIterator} to use for
252     * dividing text into passages.  This returns
253     * {@link BreakIterator#getSentenceInstance(Locale)} by default;
254     * subclasses can override to customize.
255     * <p>
256     * Note: this highlighter will call
257     * {@link BreakIterator#preceding(int)} and {@link BreakIterator#next()} many times on it.
258     * The default generic JDK implementation of {@code preceding} performs poorly.
259     */
260    protected BreakIterator getBreakIterator(String field) {
261      return defaultBreakIterator.get();
262    }
263
264    /**
265     * Returns the {@link PassageScorer} to use for
266     * ranking passages.  This
267     * returns a new {@code PassageScorer} by default;
268     * subclasses can override to customize.
269     */
270    protected PassageScorer getScorer(String field) {
271      return defaultScorer;
272    }
273
274    /**
275     * Returns the {@link PassageFormatter} to use for
276     * formatting passages into highlighted snippets.  This
277     * returns a new {@code PassageFormatter} by default;
278     * subclasses can override to customize.
279     */
280    protected PassageFormatter getFormatter(String field) {
281      return defaultFormatter;
282    }
283
284    /**
285     * Returns the number of leading passages (as delineated by the {@link BreakIterator}) when r
286     * highlights could be found.  If it's less than 0 (the default) then this defaults to the {@
287     * parameter given for each request.  If this is 0 then the resulting highlight is null (not
288     */
289    protected int getMaxNoHighlightPassages(String field) {
290      return defaultMaxNoHighlightPassages;
291    }
292
293    /**
294     * Limits the amount of field value pre-fetching until this threshold is passed.  The highlig
295     * internally highlights in batches of documents sized on the sum field value length (in char
```

```
296     * to be highlighted (bounded by {@link #getMaxLength()} for each field).  By setting this to
297     * documents to be fetched and highlighted one at a time, which you usually shouldn't do.
298     * The default is 524288 chars which translates to about a megabyte.  However, note
299     * that the highlighter sometimes ignores this and highlights one document at a time (without
300     * bunch of documents in advance) when it can detect there's no point in it -- such as when a
301     * highlighted via re-analysis as one example.
302     */
303    public int getCacheFieldValCharsThreshold() { // question: should we size by bytes instead?
304      return cacheFieldValCharsThreshold;
305    }
306
307    /**
308     * ... as passed in from constructor.
309     */
310    public IndexSearcher getIndexSearcher() {
311      return searcher;
312    }
313
314    /**
315     * ... as passed in from constructor.
316     */
317    public Analyzer getIndexAnalyzer() {
318      return indexAnalyzer;
319    }
320
321    /**
322     * Source of term offsets; essential for highlighting.
323     */
324    public enum OffsetSource {
325      POSTINGS, TERM_VECTORS, ANALYSIS, POSTINGS_WITH_TERM_VECTORS, NONE_NEEDED
326    }
327
328    /**
329     * Determine the offset source for the specified field.  The default algorithm is as follows
330     * <ol>
331     * <li>This calls {@link #getFieldInfo(String)}. Note this returns null if there is no search
332     * field isn't found there.</li>
333     * <li> If there's a field info it has
334     * {@link IndexOptions#DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS} then {@link OffsetSource#POS
335     * returned.</li>
336     * <li>If there's a field info and {@link FieldInfo#hasVectors()} then {@link OffsetSource#TE
337     * returned (note we can't check here if the TV has offsets; if there isn't then an exception
338     * down the line).</li>
339     * <li>Fall-back: {@link OffsetSource#ANALYSIS} is returned.</li>
340     * </ol>
341     * <p>
342     * Note that the highlighter sometimes switches to something else based on the query, such as
343     * {@link OffsetSource#POSTINGS_WITH_TERM_VECTORS} but in fact don't need term vectors.
344     */
345    protected OffsetSource getOffsetSource(String field) {
346      FieldInfo fieldInfo = getFieldInfo(field);
347      if (fieldInfo != null) {
```

```
348       if (fieldInfo.getIndexOptions() == IndexOptions.DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS)
349         return fieldInfo.hasVectors() ? OffsetSource.POSTINGS_WITH_TERM_VECTORS : OffsetSource
350       }
351       if (fieldInfo.hasVectors()) { // unfortunately we can't also check if the TV has offsets
352         return OffsetSource.TERM_VECTORS;
353       }
354     }
355     return OffsetSource.ANALYSIS;
356   }
357
358   /**
359    * Called by the default implementation of {@link #getOffsetSource(String)}.
360    * If there is no searcher then we simply always return null.
361    */
362   protected FieldInfo getFieldInfo(String field) {
363     if (searcher == null) {
364       return null;
365     }
366     // Need thread-safety for lazy-init but lets avoid 'synchronized' by using double-check loc
367     FieldInfos fieldInfos = this.fieldInfos; // note: it's volatile; read once
368     if (fieldInfos == null) {
369       synchronized (this) {
370         fieldInfos = this.fieldInfos;
371         if (fieldInfos == null) {
372           fieldInfos = FieldInfos.getMergedFieldInfos(searcher.getIndexReader());
373           this.fieldInfos = fieldInfos;
374         }
375
376       }
377
378     }
379     return fieldInfos.fieldInfo(field);
380   }
381
382   /**
383    * Highlights the top passages from a single field.
384    *
385    * @param field   field name to highlight.
386    *                Must have a stored string value and also be indexed with offsets.
387    * @param query   query to highlight.
388    * @param topDocs TopDocs containing the summary result documents to highlight.
389    * @return Array of formatted snippets corresponding to the documents in <code>topDocs</code>
390    * If no highlights were found for a document, the
391    * first sentence for the field will be returned.
392    * @throws IOException              if an I/O error occurred during processing
393    * @throws IllegalArgumentException if <code>field</code> was indexed without
394    *                                  {@link IndexOptions#DOCS_AND_FREQS_AND_POSITIONS_AND_OFFS
395    */
396   public String[] highlight(String field, Query query, TopDocs topDocs) throws IOException {
397     return highlight(field, query, topDocs, 1);
398   }
399
```

```
400   /**
401    * Highlights the top-N passages from a single field.
402    *
403    * @param field       field name to highlight. Must have a stored string value.
404    * @param query       query to highlight.
405    * @param topDocs     TopDocs containing the summary result documents to highlight.
406    * @param maxPassages The maximum number of top-N ranked passages used to
407    *                    form the highlighted snippets.
408    * @return Array of formatted snippets corresponding to the documents in <code>topDocs</code>
409    * If no highlights were found for a document, the
410    * first {@code maxPassages} sentences from the
411    * field will be returned.
412    * @throws IOException              if an I/O error occurred during processing
413    * @throws IllegalArgumentException if <code>field</code> was indexed without
414    *                                  {@link IndexOptions#DOCS_AND_FREQS_AND_POSITIONS_AND_OFFS
415    */
416   public String[] highlight(String field, Query query, TopDocs topDocs, int maxPassages) throws
417     Map<String, String[]> res = highlightFields(new String[]{field}, query, topDocs, new int[]{
418     return res.get(field);
419   }
420
421   /**
422    * Highlights the top passages from multiple fields.
423    * <p>
424    * Conceptually, this behaves as a more efficient form of:
425    * <pre class="prettyprint">
426    * Map m = new HashMap();
427    * for (String field : fields) {
428    * m.put(field, highlight(field, query, topDocs));
429    * }
430    * return m;
431    * </pre>
432    *
433    * @param fields  field names to highlight. Must have a stored string value.
434    * @param query   query to highlight.
435    * @param topDocs TopDocs containing the summary result documents to highlight.
436    * @return Map keyed on field name, containing the array of formatted snippets
437    * corresponding to the documents in <code>topDocs</code>.
438    * If no highlights were found for a document, the
439    * first sentence from the field will be returned.
440    * @throws IOException              if an I/O error occurred during processing
441    * @throws IllegalArgumentException if <code>field</code> was indexed without
442    *                                  {@link IndexOptions#DOCS_AND_FREQS_AND_POSITIONS_AND_OFFS
443    */
444   public Map<String, String[]> highlightFields(String[] fields, Query query, TopDocs topDocs) t
445     int maxPassages[] = new int[fields.length];
446     Arrays.fill(maxPassages, 1);
447     return highlightFields(fields, query, topDocs, maxPassages);
448   }
449
450   /**
451    * Highlights the top-N passages from multiple fields.
```

```
452     * <p>
453     * Conceptually, this behaves as a more efficient form of:
454     * <pre class="prettyprint">
455     * Map m = new HashMap();
456     * for (String field : fields) {
457     * m.put(field, highlight(field, query, topDocs, maxPassages));
458     * }
459     * return m;
460     * </pre>
461     *
462     * @param fields      field names to highlight. Must have a stored string value.
463     * @param query       query to highlight.
464     * @param topDocs     TopDocs containing the summary result documents to highlight.
465     * @param maxPassages The maximum number of top-N ranked passages per-field used to
466     *                    form the highlighted snippets.
467     * @return Map keyed on field name, containing the array of formatted snippets
468     * corresponding to the documents in <code>topDocs</code>.
469     * If no highlights were found for a document, the
470     * first {@code maxPassages} sentences from the
471     * field will be returned.
472     * @throws IOException              if an I/O error occurred during processing
473     * @throws IllegalArgumentException if <code>field</code> was indexed without
474     *                                  {@link IndexOptions#DOCS_AND_FREQS_AND_POSITIONS_AND_OFFS
475     */
476    public Map<String, String[]> highlightFields(String[] fields, Query query, TopDocs topDocs, 
477        throws IOException {
478      final ScoreDoc scoreDocs[] = topDocs.scoreDocs;
479      int docids[] = new int[scoreDocs.length];
480      for (int i = 0; i < docids.length; i++) {
481        docids[i] = scoreDocs[i].doc;
482      }
483
484      return highlightFields(fields, query, docids, maxPassages);
485    }
486
487    /**
488     * Highlights the top-N passages from multiple fields,
489     * for the provided int[] docids.
490     *
491     * @param fieldsIn      field names to highlight. Must have a stored string value.
492     * @param query         query to highlight.
493     * @param docidsIn      containing the document IDs to highlight.
494     * @param maxPassagesIn The maximum number of top-N ranked passages per-field used to
495     *                      form the highlighted snippets.
496     * @return Map keyed on field name, containing the array of formatted snippets
497     * corresponding to the documents in <code>docidsIn</code>.
498     * If no highlights were found for a document, the
499     * first {@code maxPassages} from the field will
500     * be returned.
501     * @throws IOException              if an I/O error occurred during processing
502     * @throws IllegalArgumentException if <code>field</code> was indexed without
503     *                                  {@link IndexOptions#DOCS_AND_FREQS_AND_POSITIONS_AND_OFFS
```

```
504     */
505    public Map<String, String[]> highlightFields(String[] fieldsIn, Query query, int[] docidsIn,
506        throws IOException {
507      Map<String, String[]> snippets = new HashMap<>();
508      for (Map.Entry<String, Object[]> ent : highlightFieldsAsObjects(fieldsIn, query, docidsIn,
509        Object[] snippetObjects = ent.getValue();
510        String[] snippetStrings = new String[snippetObjects.length];
511        snippets.put(ent.getKey(), snippetStrings);
512        for (int i = 0; i < snippetObjects.length; i++) {
513          Object snippet = snippetObjects[i];
514          if (snippet != null) {
515            snippetStrings[i] = snippet.toString();
516          }
517        }
518      }
519
520      return snippets;
521    }
522
523    /**
524     * Expert: highlights the top-N passages from multiple fields,
525     * for the provided int[] docids, to custom Object as
526     * returned by the {@link PassageFormatter}.  Use
527     * this API to render to something other than String.
528     *
529     * @param fieldsIn      field names to highlight. Must have a stored string value.
530     * @param query         query to highlight.
531     * @param docIdsIn      containing the document IDs to highlight.
532     * @param maxPassagesIn The maximum number of top-N ranked passages per-field used to
533     *                      form the highlighted snippets.
534     * @return Map keyed on field name, containing the array of formatted snippets
535     * corresponding to the documents in <code>docIdsIn</code>.
536     * If no highlights were found for a document, the
537     * first {@code maxPassages} from the field will
538     * be returned.
539     * @throws IOException              if an I/O error occurred during processing
540     * @throws IllegalArgumentException if <code>field</code> was indexed without
541     *                                  {@link IndexOptions#DOCS_AND_FREQS_AND_POSITIONS_AND_OFFS
542     */
543    protected Map<String, Object[]> highlightFieldsAsObjects(String[] fieldsIn, Query query, int
544                                                             int[] maxPassagesIn) throws IOExcep
545      if (fieldsIn.length < 1) {
546        throw new IllegalArgumentException("fieldsIn must not be empty");
547      }
548      if (fieldsIn.length != maxPassagesIn.length) {
549        throw new IllegalArgumentException("invalid number of maxPassagesIn");
550      }
551      if (searcher == null) {
552        throw new IllegalStateException("This method requires that an indexSearcher was passed in
553            + "constructor.  Perhaps you mean to call highlightWithoutSearcher?");
554      }
555
```

```
556       // Sort docs & fields for sequential i/o
557
558       // Sort doc IDs w/ index to original order: (copy input arrays since we sort in-place)
559       int[] docIds = new int[docIdsIn.length];
560       int[] docInIndexes = new int[docIds.length]; // fill in ascending order; points into docIds
561       copyAndSortDocIdsWithIndex(docIdsIn, docIds, docInIndexes); // latter 2 are "out" params
562
563       // Sort fields w/ maxPassages pair: (copy input arrays since we sort in-place)
564       final String fields[] = new String[fieldsIn.length];
565       final int maxPassages[] = new int[maxPassagesIn.length];
566       copyAndSortFieldsWithMaxPassages(fieldsIn, maxPassagesIn, fields, maxPassages); // latter 
567
568       // Init field highlighters (where most of the highlight logic lives, and on a per field bas
569       Set<Term> queryTerms = extractTerms(query);
570       FieldHighlighter[] fieldHighlighters = new FieldHighlighter[fields.length];
571       int numTermVectors = 0;
572       int numPostings = 0;
573       for (int f = 0; f < fields.length; f++) {
574         FieldHighlighter fieldHighlighter = getFieldHighlighter(fields[f], query, queryTerms, max
575         fieldHighlighters[f] = fieldHighlighter;
576
577         switch (fieldHighlighter.getOffsetSource()) {
578           case TERM_VECTORS:
579             numTermVectors++;
580             break;
581           case POSTINGS:
582             numPostings++;
583             break;
584           case POSTINGS_WITH_TERM_VECTORS:
585             numTermVectors++;
586             numPostings++;
587             break;
588           case ANALYSIS:
589           case NONE_NEEDED:
590           default:
591             //do nothing
592             break;
593         }
594       }
595
596       int cacheCharsThreshold = calculateOptimalCacheCharsThreshold(numTermVectors, numPostings);
597
598       IndexReader indexReaderWithTermVecCache =
599           (numTermVectors >= 2) ? TermVectorReusingLeafReader.wrap(searcher.getIndexReader()) : r
600
601       // [fieldIdx][docIdInIndex] of highlightDoc result
602       Object[][] highlightDocsInByField = new Object[fields.length][docIds.length];
603       // Highlight in doc batches determined by loadFieldValues (consumes from docIdIter)
604       DocIdSetIterator docIdIter = asDocIdSetIterator(docIds);
605       for (int batchDocIdx = 0; batchDocIdx < docIds.length; ) {
606         // Load the field values of the first batch of document(s) (note: commonly all docs are i
607         List<CharSequence[]> fieldValsByDoc =
```

```
608           loadFieldValues(fields, docIdIter, cacheCharsThreshold);
609         //    the size of the above list is the size of the batch (num of docs in the batch)
610
611         // Highlight in per-field order first, then by doc (better I/O pattern)
612         for (int fieldIdx = 0; fieldIdx < fields.length; fieldIdx++) {
613           Object[] resultByDocIn = highlightDocsInByField[fieldIdx];//parallel to docIdsIn
614           FieldHighlighter fieldHighlighter = fieldHighlighters[fieldIdx];
615           for (int docIdx = batchDocIdx; docIdx - batchDocIdx < fieldValsByDoc.size(); docIdx++)
616             int docId = docIds[docIdx];//sorted order
617             CharSequence content = fieldValsByDoc.get(docIdx - batchDocIdx)[fieldIdx];
618             if (content == null) {
619               continue;
620             }
621             IndexReader indexReader =
622                 (fieldHighlighter.getOffsetSource() == OffsetSource.TERM_VECTORS
623                     && indexReaderWithTermVecCache != null)
624                     ? indexReaderWithTermVecCache
625                     : searcher.getIndexReader();
626             final LeafReader leafReader;
627             if (indexReader instanceof LeafReader) {
628               leafReader = (LeafReader) indexReader;
629             } else {
630               List<LeafReaderContext> leaves = indexReader.leaves();
631               LeafReaderContext leafReaderContext = leaves.get(ReaderUtil.subIndex(docId, leaves)
632               leafReader = leafReaderContext.reader();
633               docId -= leafReaderContext.docBase; // adjust 'doc' to be within this leaf reader
634             }
635             int docInIndex = docInIndexes[docIdx];//original input order
636             assert resultByDocIn[docInIndex] == null;
637             resultByDocIn[docInIndex] =
638                 fieldHighlighter
639                     .highlightFieldForDoc(leafReader, docId, content.toString());
640           }
641
642         }
643
644         batchDocIdx += fieldValsByDoc.size();
645       }
646       assert docIdIter.docID() == DocIdSetIterator.NO_MORE_DOCS
647           || docIdIter.nextDoc() == DocIdSetIterator.NO_MORE_DOCS;
648
649       // TODO reconsider the return type; since this is an "advanced" method, lets not return a M
650       //    caller simply iterates it to build another structure.
651
652       // field -> object highlights parallel to docIdsIn
653       Map<String, Object[]> resultMap = new HashMap<>(fields.length);
654       for (int f = 0; f < fields.length; f++) {
655         resultMap.put(fields[f], highlightDocsInByField[f]);
656       }
657       return resultMap;
658     }
659
```

```
660   /**
661    * When cacheCharsThreshold is 0, loadFieldValues() only fetches one document at a time.  We
662    * in two circumstances:
663    */
664   private int calculateOptimalCacheCharsThreshold(int numTermVectors, int numPostings) {
665     if (numPostings == 0 && numTermVectors == 0) {
666       // (1) When all fields are ANALYSIS there's no point in caching a batch of documents
667       // because no other info on disk is needed to highlight it.
668       return 0;
669     } else if (numTermVectors >= 2) {
670       // (2) When two or more fields have term vectors, given the field-then-doc algorithm, the
671       // vectors will be fetched in a terrible access pattern unless we highlight a doc at a ti
672       // current-doc TV cache.  So we do that.  Hopefully one day TVs will be improved to make
673       return 0;
674     } else {
675       return getCacheFieldValCharsThreshold();
676     }
677   }
678
679   private void copyAndSortFieldsWithMaxPassages(String[] fieldsIn, int[] maxPassagesIn, final 
680                                                 final int[] maxPassages) {
681     System.arraycopy(fieldsIn, 0, fields, 0, fieldsIn.length);
682     System.arraycopy(maxPassagesIn, 0, maxPassages, 0, maxPassagesIn.length);
683     new InPlaceMergeSorter() {
684       @Override
685       protected void swap(int i, int j) {
686         String tmp = fields[i];
687         fields[i] = fields[j];
688         fields[j] = tmp;
689         int tmp2 = maxPassages[i];
690         maxPassages[i] = maxPassages[j];
691         maxPassages[j] = tmp2;
692       }
693
694       @Override
695       protected int compare(int i, int j) {
696         return fields[i].compareTo(fields[j]);
697       }
698
699     }.sort(0, fields.length);
700   }
701
702   private void copyAndSortDocIdsWithIndex(int[] docIdsIn, final int[] docIds, final int[] docI
703     System.arraycopy(docIdsIn, 0, docIds, 0, docIdsIn.length);
704     for (int i = 0; i < docInIndexes.length; i++) {
705       docInIndexes[i] = i;
706     }
707     new InPlaceMergeSorter() {
708       @Override
709       protected void swap(int i, int j) {
710         int tmp = docIds[i];
711         docIds[i] = docIds[j];
```

```
712         docIds[j] = tmp;
713         tmp = docInIndexes[i];
714         docInIndexes[i] = docInIndexes[j];
715         docInIndexes[j] = tmp;
716       }
717
718       @Override
719       protected int compare(int i, int j) {
720         return Integer.compare(docIds[i], docIds[j]);
721       }
722     }.sort(0, docIds.length);
723   }
724
725   /**
726    * Highlights text passed as a parameter.  This requires the {@link IndexSearcher} provided t
727    * null.  This use-case is more rare.  Naturally, the mode of operation will be {@link Offset
728    * The result of this method is whatever the {@link PassageFormatter} returns.  For the {@li
729    * DefaultPassageFormatter} and assuming {@code content} has non-zero length, the result wil
730    * string -- so it's safe to call {@link Object#toString()} on it in that case.
731    *
732    * @param field       field name to highlight (as found in the query).
733    * @param query       query to highlight.
734    * @param content     text to highlight.
735    * @param maxPassages The maximum number of top-N ranked passages used to
736    *                    form the highlighted snippets.
737    * @return result of the {@link PassageFormatter} -- probably a String.  Might be null.
738    * @throws IOException if an I/O error occurred during processing
739    */
740   //TODO make content a List? and return a List? and ensure getEmptyHighlight is never invoked
741   public Object highlightWithoutSearcher(String field, Query query, String content, int maxPass
742       throws IOException {
743     if (this.searcher != null) {
744       throw new IllegalStateException("highlightWithoutSearcher should only be called on a " +
745           getClass().getSimpleName() + " without an IndexSearcher.");
746     }
747     Objects.requireNonNull(content, "content is required");
748     Set<Term> queryTerms = extractTerms(query);
749     return getFieldHighlighter(field, query, queryTerms, maxPassages)
750         .highlightFieldForDoc(null, -1, content);
751   }
752
753   protected FieldHighlighter getFieldHighlighter(String field, Query query, Set<Term> allTerms,
754     UHComponents components = getHighlightComponents(field, query, allTerms);
755     OffsetSource offsetSource = getOptimizedOffsetSource(components);
756     return new FieldHighlighter(field,
757         getOffsetStrategy(offsetSource, components),
758         new SplittingBreakIterator(getBreakIterator(field), UnifiedHighlighter.MULTIVAL_SEP_CH
759         getScorer(field),
760         maxPassages,
761         getMaxNoHighlightPassages(field),
762         getFormatter(field));
763   }
```

```
764
765   protected UHComponents getHighlightComponents(String field, Query query, Set<Term> allTerms)
766     Predicate<String> fieldMatcher = getFieldMatcher(field);
767     Set<HighlightFlag> highlightFlags = getFlags(field);
768     PhraseHelper phraseHelper = getPhraseHelper(field, query, highlightFlags);
769     boolean queryHasUnrecognizedPart = hasUnrecognizedQuery(fieldMatcher, query);
770     BytesRef[] terms = null;
771     LabelledCharArrayMatcher[] automata = null;
772     if (!highlightFlags.contains(HighlightFlag.WEIGHT_MATCHES) || !queryHasUnrecognizedPart) {
773       terms = filterExtractedTerms(fieldMatcher, allTerms);
774       automata = getAutomata(field, query, highlightFlags);
775     } // otherwise don't need to extract
776     return new UHComponents(field, fieldMatcher, query, terms, phraseHelper, automata, queryHas
777   }
778
779   protected boolean hasUnrecognizedQuery(Predicate<String> fieldMatcher, Query query) {
780     boolean[] hasUnknownLeaf = new boolean[1];
781     query.visit(new QueryVisitor() {
782       @Override
783       public boolean acceptField(String field) {
784         // checking hasUnknownLeaf is a trick to exit early
785         return hasUnknownLeaf[0] == false && fieldMatcher.test(field);
786       }
787
788       @Override
789       public void visitLeaf(Query query) {
790         if (MultiTermHighlighting.canExtractAutomataFromLeafQuery(query) == false) {
791           if (!(query instanceof MatchAllDocsQuery || query instanceof MatchNoDocsQuery)) {
792             hasUnknownLeaf[0] = true;
793           }
794         }
795       }
796     });
797     return hasUnknownLeaf[0];
798   }
799
800   protected static BytesRef[] filterExtractedTerms(Predicate<String> fieldMatcher, Set<Term> qu
801     // Strip off the redundant field and sort the remaining terms
802     SortedSet<BytesRef> filteredTerms = new TreeSet<>();
803     for (Term term : queryTerms) {
804       if (fieldMatcher.test(term.field())) {
805         filteredTerms.add(term.bytes());
806       }
807     }
808     return filteredTerms.toArray(new BytesRef[filteredTerms.size()]);
809   }
810
811   protected Set<HighlightFlag> getFlags(String field) {
812     Set<HighlightFlag> highlightFlags = EnumSet.noneOf(HighlightFlag.class);
813     if (shouldHandleMultiTermQuery(field)) {
814       highlightFlags.add(HighlightFlag.MULTI_TERM_QUERY);
815     }
```

```
816       if (shouldHighlightPhrasesStrictly(field)) {
817         highlightFlags.add(HighlightFlag.PHRASES);
818       }
819       if (shouldPreferPassageRelevancyOverSpeed(field)) {
820         highlightFlags.add(HighlightFlag.PASSAGE_RELEVANCY_OVER_SPEED);
821       }
822       return highlightFlags;
823     }
824
825     protected PhraseHelper getPhraseHelper(String field, Query query, Set<HighlightFlag> highligh
826       boolean useWeightMatchesIter = highlightFlags.contains(HighlightFlag.WEIGHT_MATCHES);
827       if (useWeightMatchesIter) {
828         return PhraseHelper.NONE; // will be handled by Weight.matches which always considers phr
829       }
830       boolean highlightPhrasesStrictly = highlightFlags.contains(HighlightFlag.PHRASES);
831       boolean handleMultiTermQuery = highlightFlags.contains(HighlightFlag.MULTI_TERM_QUERY);
832       return highlightPhrasesStrictly ?
833           new PhraseHelper(query, field, getFieldMatcher(field),
834               this::requiresRewrite,
835               this::preSpanQueryRewrite,
836               !handleMultiTermQuery
837           )
838           : PhraseHelper.NONE;
839     }
840
841     protected LabelledCharArrayMatcher[] getAutomata(String field, Query query, Set<HighlightFlag
842       // do we "eagerly" look in span queries for automata here, or do we not and let PhraseHelpe
843       // if don't highlight phrases strictly,
844       final boolean lookInSpan =
845           !highlightFlags.contains(HighlightFlag.PHRASES) // no PhraseHelper
846           || highlightFlags.contains(HighlightFlag.WEIGHT_MATCHES); // Weight.Matches will find a
847
848       return highlightFlags.contains(HighlightFlag.MULTI_TERM_QUERY)
849           ? MultiTermHighlighting.extractAutomata(query, getFieldMatcher(field), lookInSpan)
850           : ZERO_LEN_AUTOMATA_ARRAY;
851     }
852
853     protected OffsetSource getOptimizedOffsetSource(UHComponents components) {
854       OffsetSource offsetSource = getOffsetSource(components.getField());
855
856       // null automata means unknown, so assume a possibility
857       boolean mtqOrRewrite = components.getAutomata() == null || components.getAutomata().length
858           || components.getPhraseHelper().willRewrite() || components.hasUnrecognizedQueryPart()
859
860       // null terms means unknown, so assume something to highlight
861       if (mtqOrRewrite == false && components.getTerms() != null && components.getTerms().length
862         return OffsetSource.NONE_NEEDED; //nothing to highlight
863       }
864
865       switch (offsetSource) {
866         case POSTINGS:
867           if (mtqOrRewrite) { // may need to see scan through all terms for the highlighted docu
```

```
868           return OffsetSource.ANALYSIS;
869         }
870         break;
871       case POSTINGS_WITH_TERM_VECTORS:
872         if (mtqOrRewrite == false) {
873           return OffsetSource.POSTINGS; //We don't need term vectors
874         }
875         break;
876       case ANALYSIS:
877       case TERM_VECTORS:
878       case NONE_NEEDED:
879       default:
880         //stick with the original offset source
881         break;
882     }
883
884     return offsetSource;
885   }
886
887   protected FieldOffsetStrategy getOffsetStrategy(OffsetSource offsetSource, UHComponents compo
888     switch (offsetSource) {
889       case ANALYSIS:
890         if (!components.getPhraseHelper().hasPositionSensitivity() &&
891             !components.getHighlightFlags().contains(HighlightFlag.PASSAGE_RELEVANCY_OVER_SPEED
892             !components.getHighlightFlags().contains(HighlightFlag.WEIGHT_MATCHES)) {
893           //skip using a memory index since it's pure term filtering
894           return new TokenStreamOffsetStrategy(components, getIndexAnalyzer());
895         } else {
896           return new MemoryIndexOffsetStrategy(components, getIndexAnalyzer());
897         }
898       case NONE_NEEDED:
899         return NoOpOffsetStrategy.INSTANCE;
900       case TERM_VECTORS:
901         return new TermVectorOffsetStrategy(components);
902       case POSTINGS:
903         return new PostingsOffsetStrategy(components);
904       case POSTINGS_WITH_TERM_VECTORS:
905         return new PostingsWithTermVectorsOffsetStrategy(components);
906       default:
907         throw new IllegalArgumentException("Unrecognized offset source " + offsetSource);
908     }
909   }
910
911   /**
912    * When highlighting phrases accurately, we need to know which {@link SpanQuery}'s need to ha
913    * {@link Query#rewrite(IndexReader)} called on them.  It helps performance to avoid it if it
914    * This method will be invoked on all SpanQuery instances recursively. If you have custom Spa
915    * override this to check instanceof and provide a definitive answer. If the query isn't your
916    * return null to have the default rules apply, which govern the ones included in Lucene.
917    */
918   protected Boolean requiresRewrite(SpanQuery spanQuery) {
919     return null;
```

```
920   }
921
922   /**
923    * When highlighting phrases accurately, we may need to handle custom queries that aren't sup
924    * {@link org.apache.lucene.search.highlight.WeightedSpanTermExtractor} as called by the {@c
925    * Should custom query types be needed, this method should be overriden to return a collectio
926    * or null if nothing to do. If the query is not custom, simply returning null will allow the
927    *
928    * @param query Query to be highlighted
929    * @return A Collection of Query object(s) if needs to be rewritten, otherwise null.
930    */
931   protected Collection<Query> preSpanQueryRewrite(Query query) {
932     return null;
933   }
934
935   private DocIdSetIterator asDocIdSetIterator(int[] sortedDocIds) {
936     return new DocIdSetIterator() {
937       int idx = -1;
938
939       @Override
940       public int docID() {
941         if (idx < 0 || idx >= sortedDocIds.length) {
942           return NO_MORE_DOCS;
943         }
944         return sortedDocIds[idx];
945       }
946
947       @Override
948       public int nextDoc() throws IOException {
949         idx++;
950         return docID();
951       }
952
953       @Override
954       public int advance(int target) throws IOException {
955         return super.slowAdvance(target); // won't be called, so whatever
956       }
957
958       @Override
959       public long cost() {
960         return Math.max(0, sortedDocIds.length - (idx + 1)); // remaining docs
961       }
962     };
963   }
964
965   /**
966    * Loads the String values for each docId by field to be highlighted.  By default this loads
967    * by the same name as given, but a subclass can change the source.  The returned Strings mus
968    * what was indexed (at least for postings or term-vectors offset sources).
969    * This method must load fields for at least one document from the given {@link DocIdSetItera
970    * but need not return all of them; by default the character lengths are summed and this meth
971    * when {@code cacheCharsThreshold} is exceeded.  Specifically if that number is 0, then only
```

```
   * fetched no matter what.  Values in the array of {@link CharSequence} will be null if no va
   */
  protected List<CharSequence[]> loadFieldValues(String[] fields,
                                                 DocIdSetIterator docIter, int cacheCharsThresh
      throws IOException {
    List<CharSequence[]> docListOfFields =
        new ArrayList<>(cacheCharsThreshold == 0 ? 1 : (int) Math.min(64, docIter.cost()));

    LimitedStoredFieldVisitor visitor = newLimitedStoredFieldsVisitor(fields);
    int sumChars = 0;
    do {
      int docId = docIter.nextDoc();
      if (docId == DocIdSetIterator.NO_MORE_DOCS) {
        break;
      }
      visitor.init();
      searcher.doc(docId, visitor);
      CharSequence[] valuesByField = visitor.getValuesByField();
      docListOfFields.add(valuesByField);
      for (CharSequence val : valuesByField) {
        sumChars += (val == null ? 0 : val.length());
      }
    } while (sumChars <= cacheCharsThreshold && cacheCharsThreshold != 0);
    return docListOfFields;
  }

  /**
   * @lucene.internal
   */
  protected LimitedStoredFieldVisitor newLimitedStoredFieldsVisitor(String[] fields) {
    return new LimitedStoredFieldVisitor(fields, MULTIVAL_SEP_CHAR, getMaxLength());
  }

  /**
   * Fetches stored fields for highlighting. Uses a multi-val separator char and honors a max l
   * @lucene.internal
   */
  protected static class LimitedStoredFieldVisitor extends StoredFieldVisitor {
    protected final String[] fields;
    protected final char valueSeparator;
    protected final int maxLength;
    protected CharSequence[] values;// starts off as String; may become StringBuilder.
    protected int currentField;

    public LimitedStoredFieldVisitor(String[] fields, char valueSeparator, int maxLength) {
      this.fields = fields;
      this.valueSeparator = valueSeparator;
      this.maxLength = maxLength;
    }

    void init() {
      values = new CharSequence[fields.length];
```

```
1024        currentField = -1;
1025      }
1026
1027      @Override
1028      public void stringField(FieldInfo fieldInfo, String value) throws IOException {
1029        assert currentField >= 0;
1030        Objects.requireNonNull(value, "String value should not be null");
1031        CharSequence curValue = values[currentField];
1032        if (curValue == null) {
1033          //question: if truncate due to maxLength, should we try and avoid keeping the other cha
1034          //  the backing char[]?
1035          values[currentField] = value.substring(0, Math.min(maxLength, value.length()));//note:
1036          return;
1037        }
1038        final int lengthBudget = maxLength - curValue.length();
1039        if (lengthBudget <= 0) {
1040          return;
1041        }
1042        StringBuilder curValueBuilder;
1043        if (curValue instanceof StringBuilder) {
1044          curValueBuilder = (StringBuilder) curValue;
1045        } else {
1046          // upgrade String to StringBuilder. Choose a good initial size.
1047          curValueBuilder = new StringBuilder(curValue.length() + Math.min(lengthBudget, value.le
1048          curValueBuilder.append(curValue);
1049        }
1050        curValueBuilder.append(valueSeparator);
1051        curValueBuilder.append(value.substring(0, Math.min(lengthBudget - 1, value.length())));
1052        values[currentField] = curValueBuilder;
1053      }
1054
1055      @Override
1056      public Status needsField(FieldInfo fieldInfo) throws IOException {
1057        currentField = Arrays.binarySearch(fields, fieldInfo.name);
1058        if (currentField < 0) {
1059          return Status.NO;
1060        }
1061        CharSequence curVal = values[currentField];
1062        if (curVal != null && curVal.length() >= maxLength) {
1063          return fields.length == 1 ? Status.STOP : Status.NO;
1064        }
1065        return Status.YES;
1066      }
1067
1068      CharSequence[] getValuesByField() {
1069        return this.values;
1070      }
1071
1072    }
1073
1074    /**
1075     * Wraps an IndexReader that remembers/caches the last call to {@link LeafReader#getTermVecto
```

```
1076     * if the next call has the same ID, then it is reused.  If TV's were column-stride (like doc
1077     * be no need for this.
1078     */
1079    private static class TermVectorReusingLeafReader extends FilterLeafReader {
1080
1081      static IndexReader wrap(IndexReader reader) throws IOException {
1082        LeafReader[] leafReaders = reader.leaves().stream()
1083            .map(LeafReaderContext::reader)
1084            .map(TermVectorReusingLeafReader::new)
1085            .toArray(LeafReader[]::new);
1086        return new BaseCompositeReader<IndexReader>(leafReaders) {
1087          @Override
1088          protected void doClose() throws IOException {
1089            reader.close();
1090          }
1091
1092          @Override
1093          public CacheHelper getReaderCacheHelper() {
1094            return null;
1095          }
1096        };
1097      }
1098
1099      private int lastDocId = -1;
1100      private Fields tvFields;
1101
1102      TermVectorReusingLeafReader(LeafReader in) {
1103        super(in);
1104      }
1105
1106      @Override
1107      public Fields getTermVectors(int docID) throws IOException {
1108        if (docID != lastDocId) {
1109          lastDocId = docID;
1110          tvFields = in.getTermVectors(docID);
1111        }
1112        return tvFields;
1113      }
1114
1115      @Override
1116      public CacheHelper getCoreCacheHelper() {
1117        return null;
1118      }
1119
1120      @Override
1121      public CacheHelper getReaderCacheHelper() {
1122        return null;
1123      }
1124
1125    }
1126
1127    /**
```

```
1128     * Flags for controlling highlighting behavior.
1129     */
1130    public enum HighlightFlag {
1131      /** @see UnifiedHighlighter#setHighlightPhrasesStrictly(boolean) */
1132      PHRASES,
1133
1134      /** @see UnifiedHighlighter#setHandleMultiTermQuery(boolean) */
1135      MULTI_TERM_QUERY,
1136
1137      /** Passage relevancy is more important than speed.  True by default. */
1138      PASSAGE_RELEVANCY_OVER_SPEED,
1139
1140      /**
1141       * Internally use the {@link Weight#matches(LeafReaderContext, int)} API for highlighting.
1142       * It's more accurate to the query, though might not calculate passage relevancy as well.
1143       * Use of this flag requires {@link #MULTI_TERM_QUERY} and {@link #PHRASES}.
1144       * {@link #PASSAGE_RELEVANCY_OVER_SPEED} will be ignored.  False by default.
1145       */
1146      WEIGHT_MATCHES
1147
1148      // TODO: useQueryBoosts
1149    }
1150  }
```