Apache Lucene – Block-Max Wand Algorithm

Overview:

Apache Lucene is open source high-performance, full featured text search engine library. After its initial release in 1999, many improvements were done to the library by the open source community. In this article, I'm going to discuss the Apache Lucene Boolean query performance improvements using **Block-Max** indexes and **Block-Max WAND** algorithm.

Before we jump on to the basics of the algorithm, a small brief about the query processing techniques that are in use to tackle the longer processing times of queries.

Early termination is one important technique that addresses the above problem. We say that a query processing algorithm is exhaustive if it fully evaluates all documents that satisfy the Boolean filter condition. Any nonexhaustive algorithm is considered to use early termination (ET). There are four ways in which early termination often happens:

- **Stop early:** In this case, the postings are usually arranged such that the most promising documents appear early. Then we stop the traversal of the index as soon as we (may) have the top-k results. Well-known examples are the TA, FA, and NRA algorithms of Fagin.

- **Skip within lists:** When the postings in each list are sorted by docIDs, the promising documents are spread out throughout the inverted lists, and thus the standard intuition for "stop early" does not apply. There are few published works on early termination techniques under this scenario. **An exception is the WAND algorithm**, which uses a smart pointer movement technique to skip many documents that would be evaluated by an exhaustive algorithm.

- **Omit lists:** One or more lists for the query terms are completely ignored, if they do not affect the final results by much.
- **Score only partially:** We partially evaluate a document by computing only some term scores, or by computing approximate scores. When we find that the document cannot be in the top results, we stop evaluation.

Apache lucene uses inverted indexes for faster query processing. Each inverted index is a cluster of Lists Lw, where Lw represents a inverted list consisting of ***term w and its occurences, frequency in a given document, along with docid***.

A posting is nothing but an entry in the inverted list that can store information about a document such as docid, term frequency, page rank score, similarity function used etc.

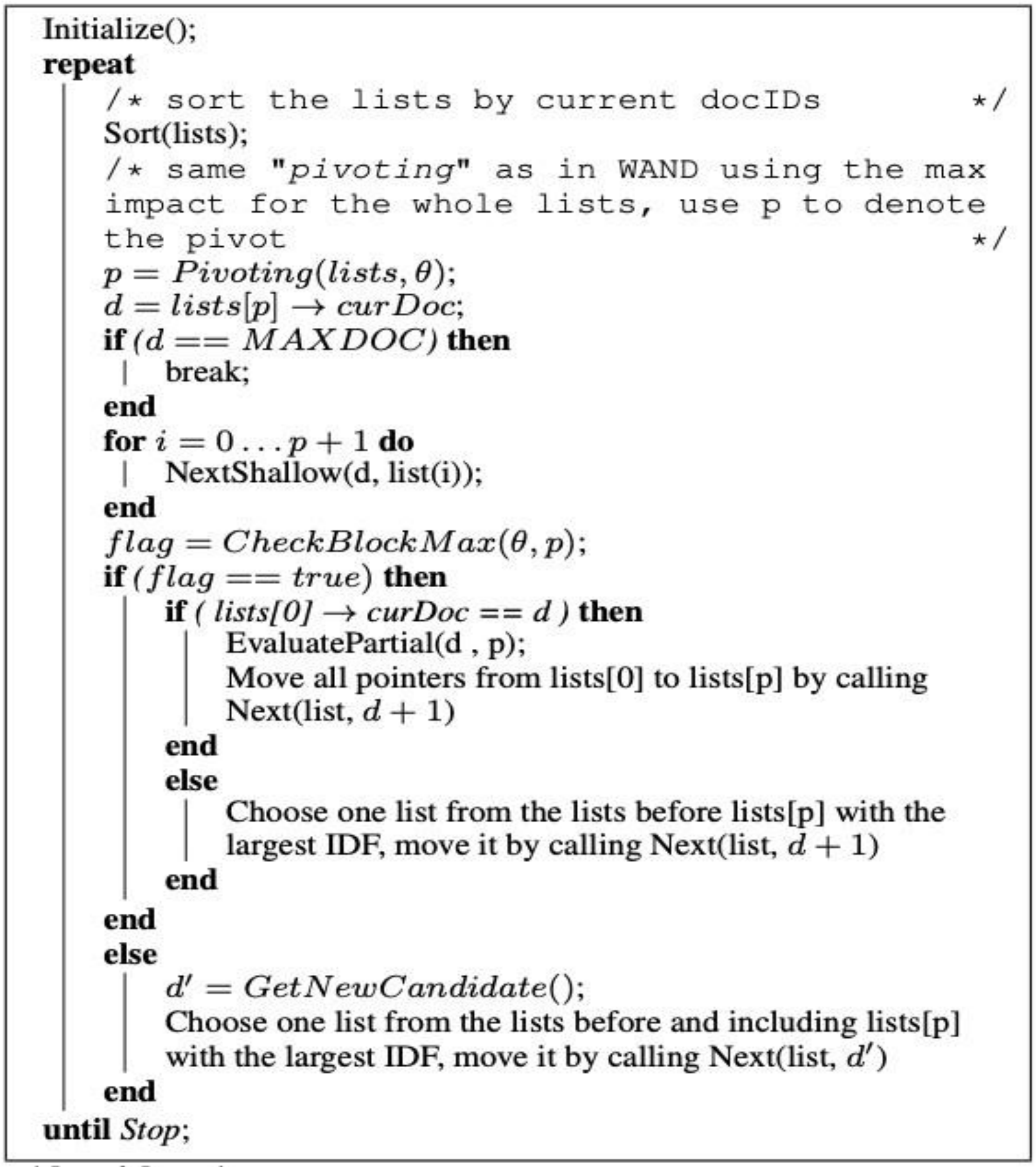
The two index traversal techniques that we use in IR realm are:

Document-At-A-Time (DAAT): In DAAT query processing, each list has a pointer that points to a "current" posting in the list. All the pointers move forward in parallel as the query is being processed.

Term-At-A-Time (TAAT): In TAAT query processing, we first access one term, or one layer from one term, and then move to the next term, or the next layer from the same term or a different term. We use a temporary data structure to keep track of currently active top-k candidates.

WAND algorithm mainly uses DAAT as it consumes less space and uses simple data structures. The algorithm primarily makes use of the global impact score defined on inverted lists and is less efficient when it comes to skipping the documents.

Below is the algorithm for the proposed by “***Shuai Ding***” and “***Torsten Suel***”:



```
Initialize();
repeat
    /* sort the lists by current docIDs */
    Sort(lists);
    /* same "pivoting" as in WAND using the max impact for the whole lists, use p to denote the pivot */
    $p = Pivoting(lists, \theta)$;
    $d = lists[p] \rightarrow curDoc$;
    if ($d == MAXDOC$) then
        break;
    end
    for $i = 0 \ldots p + 1$ do
        NextShallow(d, list(i));
    end
    $flag = CheckBlockMax(\theta, p)$;
    if ($flag == true$) then
        if ( $lists[0] \rightarrow curDoc == d$ ) then
            EvaluatePartial(d , p);
            Move all pointers from lists[0] to lists[p] by calling
            Next(list, $d + 1$)
        end
        else
            Choose one list from the lists before lists[p] with the
            largest IDF, move it by calling Next(list, $d + 1$)
        end
    end
    else
        $d' = GetNewCandidate()$;
        Choose one list from the lists before and including lists[p]
        with the largest IDF, move it by calling Next(list, $d'$)
    end
until Stop;
```

Apache Lucene – Block-Max Wand Algorithm

```
while $did > list-> blockboundary[current\_block]$ do
  | $current\_block++$;
end
```

Algorithm 2: NextShallow(list, did)

```
$maxposs = 0.0f$;
for $i = 0 \dots pivot + 1$ do
  | $maxposs+ = list[i]-> blockmax[current\_block]$;
end
if ( $maxposs > threshold$ ) then
  | return true;
end
else
  | return false;
end
```

Algorithm 3: CheckBlockMax(threshold, pivot)

Block Max Wand Algorithm is an extension to WAND algorithm, where we compute max impact score on block postings. We are splitting the inverted list into blocks of size 64 or 128 postings and calculate its max impact score. The reason we are splitting the postings into multiple small blocks, is the average impact score of an inverted list may not accurately represent the documents present in it. Hence, by splitting into smaller chunks we can estimate the impact score more accurately. A global score is an additional information about a document such as its length, pagerank etc.

Algorithm explained:

Input: Theta -> Threshold value, inverted lists.

Steps:

1. Sort the inverted lists based on the docid.
2. Get the pivot based on the max impact score of the lists and Threshold value.
3. Once we have the pivot, get the current document d.
4. **For all lists do**: Check if we can skip processing of any blocks by comparing the block boundary values with the **doc id**. ***NextShallow*** *method is used to do the same in the algorithm.*
5. CheckBlockMax is used to calculate the block maximum value by adding up the block max values of all the lists from 0 to till pivot+1 and return true if the value is greater than threshold value.

6. If true, we process evaluatepartial method and then call function NexT(list,d+1). This function returns the document in the list i, whose value is greater than or equal to d. Here, we decompress the block data in order to find out the doc id value that needs to be returned.
7. We also got a function to choose a new document in case if CheckBlockMax returns false.

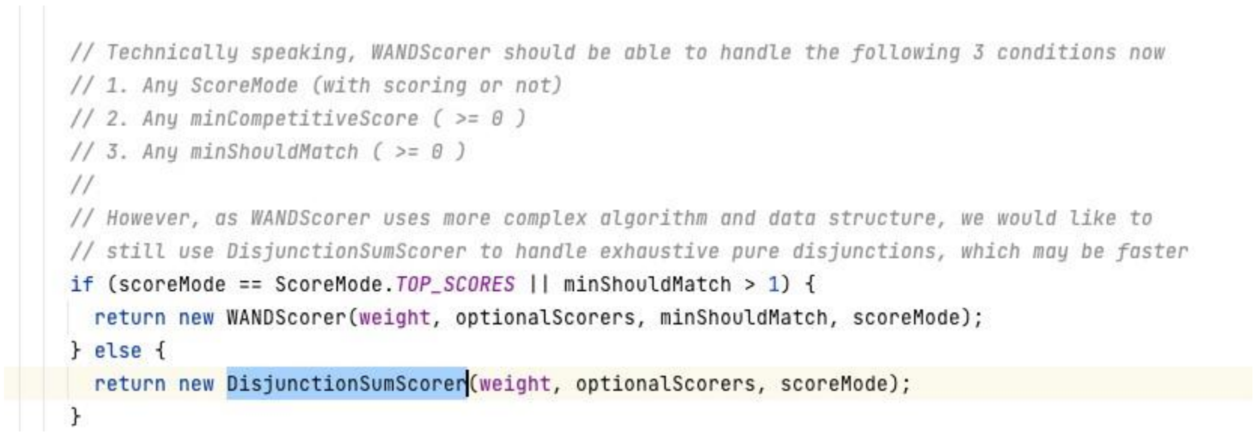
Lucene Implementation:

In apache Lucene, a ***scorer*** is an abstract class defined to expose the below functionalities:

- Expose an iterator() over documents matching a query in increasing order of doc Id.
- Compute Document scores using a given Similarity implementation.

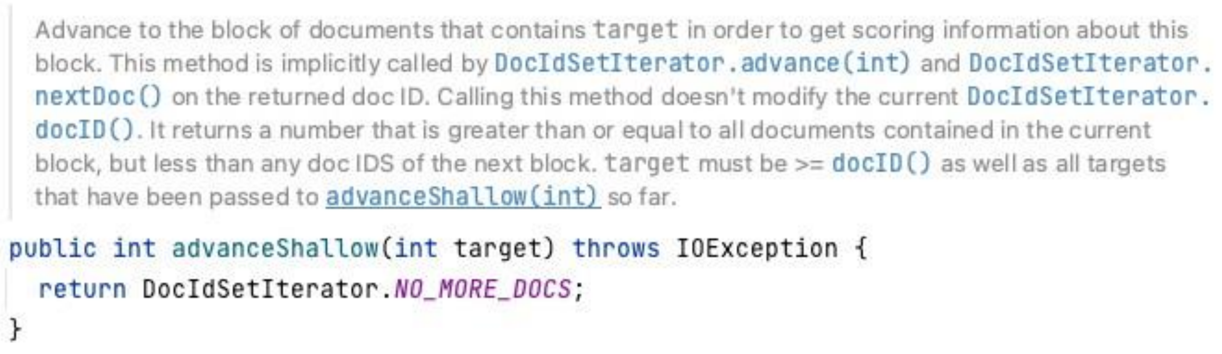
DocIdSetIterator class defines methods to iterate over a set of nondecreasing doc ids. The max doc ids value is set to 2147483647 in order to be used as a sentinel object.

Classes BlockMaxConjunctionScorer, WANDScorer and DisjunctionSumScorer implements the ***scorer*** interface and make use of the ***DocIdSetIterator*** in order to access the documents.



```
// Technically speaking, WANDScorer should be able to handle the following 3 conditions now
// 1. Any ScoreMode (with scoring or not)
// 2. Any minCompetitiveScore ( >= 0 )
// 3. Any minShouldMatch ( >= 0 )
//
// However, as WANDScorer uses more complex algorithm and data structure, we would like to
// still use DisjunctionSumScorer to handle exhaustive pure disjunctions, which may be faster
if (scoreMode == ScoreMode.TOP_SCORES || minShouldMatch > 1) {
  return new WANDScorer(weight, optionalScorers, minShouldMatch, scoreMode);
} else {
  return new DisjunctionSumScorer(weight, optionalScorers, scoreMode);
}
```

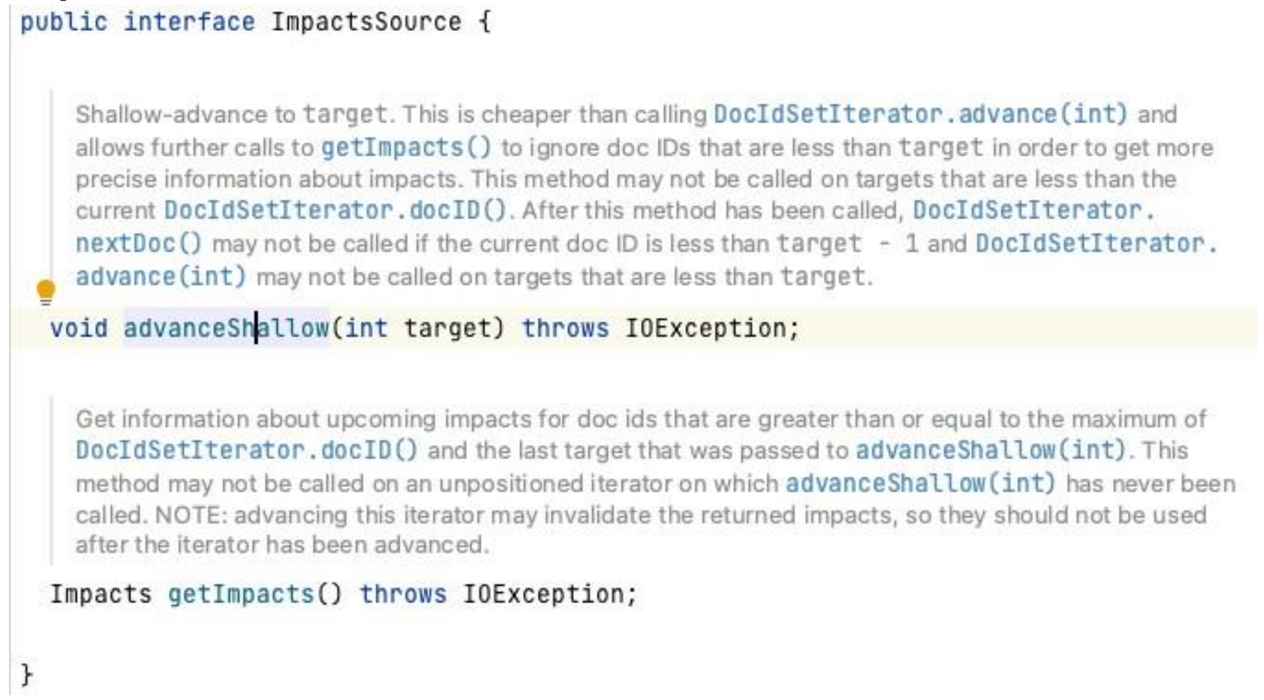
Disjunctive queries(queries with OR operator) make use of ***WANDScorer*** or ***DisJunctionSumScorer*** whereas Conjunctive queries(queries with AND operator) make use of ***BlockMaxConjunctionScorer.***



Advance to the block of documents that contains `target` in order to get scoring information about this block. This method is implicitly called by `DocIdSetIterator.advance(int)` and `DocIdSetIterator.nextDoc()` on the returned doc ID. Calling this method doesn't modify the current `DocIdSetIterator.docID()`. It returns a number that is greater than or equal to all documents contained in the current block, but less than any doc IDS of the next block. `target` must be >= `docID()` as well as all targets that have been passed to `advanceShallow(int)` so far.

```java
public int advanceShallow(int target) throws IOException {
  return DocIdSetIterator.NO_MORE_DOCS;
}
```

As described in the above algorithm, the shallow, method has been implemented in Lucene but with a small change, instead of storing the block score value in the index table, Lucene is storing the term frequency and document id, using the two we can calculate the block index value.



```java
public interface ImpactsSource {

  /**
   * Shallow-advance to target. This is cheaper than calling DocIdSetIterator.advance(int) and
   * allows further calls to getImpacts() to ignore doc IDs that are less than target in order to get more
   * precise information about impacts. This method may not be called on targets that are less than the
   * current DocIdSetIterator.docID(). After this method has been called, DocIdSetIterator.
   * nextDoc() may not be called if the current doc ID is less than target - 1 and DocIdSetIterator.
   * advance(int) may not be called on targets that are less than target.
   */
  void advanceShallow(int target) throws IOException;

  /**
   * Get information about upcoming impacts for doc ids that are greater than or equal to the maximum of
   * DocIdSetIterator.docID() and the last target that was passed to advanceShallow(int). This
   * method may not be called on an unpositioned iterator on which advanceShallow(int) has never been
   * called. NOTE: advancing this iterator may invalidate the returned impacts, so they should not be used
   * after the iterator has been advanced.
   */
  Impacts getImpacts() throws IOException;

}
```

The score functions are making use of the Lucene84PostingsFormat class that is responsible for compressing and decompressing the index data. It has also implemented functions such as ***advanceShallow(int target)*** and ***getImpacts()*** to further reduce the burden on the ***DocIdSetIterator.***
With the performance and optimization benefits, we also must compromise on some of the other stuff, such as accurate hit counts. Using this we cannot get the accurate hit counts and in order to get that we will have hit on performance.

Conclusion:

Apache Lucene – Block-Max Wand Algorithm

The implementation resulted in term queries run between 3x and 7x faster, conjunctions between 3% and 7x faster and disjunctions between -8% (slightly slower) and 15x faster when running Lucene's benchmark suite.

References:

1. http://engineering.nyu.edu/~suel/papers/bmw.pdf
2. https://lucene.apache.org/core/
3. https://www.elastic.co/blog/faster-retrieval-of-top-hits-inelasticsearch-with-block-max-wand