Querying a Lucene Index

Queries and Scorers and Weights, oh my!

Alan Woodward - alan@flax.co.uk - @romseygeek

flax
The Open Source Search Specialists

- We build, tune and support fast, accurate and highly scalable search, analytics and Big Data applications
- We use (and create) **open source** software
- We're independent, honest, and have 15+ years experience
- We also:
  - Run and attend events, meetups and conferences
  - Write extensively about search and related matters
  - Offer training and mentoring

How does a lucene query work?

- Tour through lucene classes
- Matching
- Collection
- Some queries
- Cacheing

flax
The Open Source Search Specialists

Why should I care?

IndexReader

LeafReaderContext

LeafCollector

IndexSearcher

Scorer

Weight

Collector

Query

TopDocs

IndexReader

- Mediates read-only access to the data structures of a lucene index

IndexReader

- Mediates read-only access to the data structures of a lucene index

IndexSearcher

- Wraps an IndexReader and provides methods for querying

Query

- Defines **what should be retrieved** from an index
- IndexReader **independent**
- Generally **immutable**
- Many different types shipped with lucene

TermQuery

WildcardQuery

PointRangeQuery

BooleanQuery

PhraseQuery

flax
The Open Source Search Specialists

Weight

- Representation of a Query for a **specific IndexReader**
- Not normally seen by the client
- Maintains state for a query that relates to the whole index

Weight

- Created by Query.createWeight(IndexSearcher, boolean, float)

- Not all queries can create a Weight - some need to be rewritten first

- e.g. AutomatonQuery gets rewritten against the terms dictionary to a disjunction query of some kind

A diversion…

Lucene index structure

- Indexes consist of multiple immutable **segments**
- Each segment is a mini-index
- Segments are built in memory and flushed to disk on commits
- Background merges ensure that the number of segments is kept under control

Lucene index structure

- A top-level **IndexReader** has a leaves() method that returns a list of **LeafReaderContext** objects
- Each **LeafReaderContext** records its position within the index as a whole, enabling consumers to map doc ids within the segment to an index-global id
- The **LeafReaderContext** also allows access to a **LeafReader**

What does this mean for searching?

- IndexReader only gives us a top-level view of the index and access to some statistics

- To access data structures we need to iterate over a set of LeafReader objects, one per segment

- Weight is a top-level object against an IndexReader

- We need a different object for LeafReaders

Scorer

- Maintains state for a query per LeafReader
- Provides an iterator over documents in a single segment that match the parent query
- Also provides access to the scoring mechanism
- Generated by Weight.scorer(LeafReaderContext)
- Returning a null scorer means no matches in this segment

Let’s tie it all together

flax
The Open Source Search Specialists

- **Query** objects are independent of the index
- Given an IndexReader, a **Query** can create a **Weight**
- To match documents, a **Weight** will create a **Scorer** for each segment in the index
- Each **Scorer** then provides an iterator which iterates over the matching documents in a segment

Or, in pseudo-code…

```
Weight w = query.createWeight(searcher, true, 1.0);
for (LeafReaderContext ctx: reader.leaves()) {
    Scorer s = w.scorer(ctx);
    DocIdSetIterator it = s.iterator();
    while (it.nextDoc() != NO_MORE_DOCUMENTS) {
        // .. do something with it.docId()
    }
}
```

Or, in pseudo-code…

```
Weight w = query.createWeight(searcher, true, 1.0);
for (LeafReaderContext ctx: reader.leaves()) {
    Scorer s = w.scorer(ctx);
    DocIdSetIterator it = s.iterator();
    while (it.nextDoc() != NO_MORE_DOCUMENTS) {
        // .. do something with it.docId()
    }
}
```

What do we do here?

Collector

- Defines what to do with each match as it is reached
- Top-level **Collector** has a method which returns a **LeafCollector** for each segment
- For each matching document, the LeafCollector's collect(int doc) method is called

Or, in pseudo-code…

```
Weight w = query.createWeight(searcher, true, 1.0);
for (LeafReaderContext ctx: reader.leaves()) {
    Scorer s = w.scorer(ctx);
    LeafCollector c = collector.getLeafCollector(ctx);
    c.setScorer(s);
    DocIdSetIterator it = s.iterator();
    while (it.nextDoc() != NO_MORE_DOCUMENTS) {
        c.collect(it.docID());
     }
}
```

- Lucene comes with a number of pre-packaged Collectors

- IndexSearcher.search(Query, int) uses **TopScoreDocCollector** to return the top-n matching documents, sorted by score

- IndexSearcher.search(Query, int, Sort) uses **TopFieldCollector** to return the top-n matching documents, sorted by field

- Or you can pass your own to IndexSearcher.search(Query, Collector)

- The **Top*Collector** classes use a priority queue to store their top-n hits

- Expensive for deep paging, as you need to allocate a queue that's as big as your page depth

- IndexSearcher.searchAfter(ScoreDoc, Query, int) to the rescue!

- Allows the PQ to exclude documents at the top of the queue as well as the bottom

- Collection and scoring are done at iteration time
- This means that the scoring algorithm doesn't know how many documents will match when scores are calculated
- It also doesn't know **anything** about other matching documents

- **Rescorer** allows you to run a first-pass search with a low cost scoring algorithm, and then run a second pass over the top-k results

flax
The Open Source Search Specialists

Matching

TermQuery

- Scorer implementation is **TermScorer**
- Takes a **PostingsEnum** iterator generated from a **LeafReader** via a **Terms** reference
- nextDoc() just delegates to the **PostingsEnum**
- If the **PostingsEnum** is null, then TermWeight.scorer() will also return null

BooleanQuery

- Number of different Scorer implementations depending on the clauses
- ConjunctionScorer for pure conjunctions
- DisjunctionSumScorer for pure disjunctions
- ReqOptScorer for combinations
- ReqExclScorer for exclusions

BooleanQuery

- ConjunctionScorer sorts its child scorers by their cost

- Calls nextDoc() on its lead scorer, and then advances all other scorers to the lead docId

- If it's a match, then return; otherwise, advance the lead scorer to the maximum docId of the child scorers

BooleanQuery

- DisjunctionSumScorer maintains a priority queue of its child scorers
- All scorers are advanced to their first matching document before iteration begins
- nextDoc() advances the scorer with the lowest doc id and updates the priority queue
- current docId is the docId of the bottom of the queue

BooleanQuery

- ReqOptScorer combines a conjunction and a disjunction
- If scores aren't required, it just delegates to the conjunction
- Otherwise it advances using the conjunction, and then advances the disjunction to the current doc for scoring.

BooleanQuery

- ReqExclScorer takes a child scorer of any kind (conjunction, disjunction, ReqOptScorer) and an exclusion scorer

- Advances using the child scorer, and then checks that the exclusion scorer doesn't match on the same document

PhraseQuery

- Two Scorer implementations: **ExactPhraseScorer** and **SloppyPhraseScorer**

- Take a **PostingsEnum** per term, and an offset

- nextDoc() finds the next document containing all terms, and then checks positions to see if the phrase exists

Cacheing

Caching

- Useful to cache the result of complex queries, particularly when you're not interested in scores
- IndexSearcher comes with a built-in **QueryCache** that will handle this for you

Cacheing

- Rather than calling Query.createWeight() directly, we call IndexSearcher.createWeight(Query, boolean, float)

- If scores aren't required, then the searcher's query cache will wrap the returned weight with a CacheingWrapperWeight

- This then caches the results from individual segments

Cacheing

- When Weight.scorer() is called, the CacheingWrapperWeight checks its cache to see if it can just replay the cached bitset.

- Because the cache operates at the segment level, you can re-use it when you reopen a searcher.

Caching

- How do you tell a searcher that scoring isn't required?
- Collector.needsScores()
- BooleanQuery.FILTER

flax
The Open Source Search Specialists

Questions?

flax
The Open Source Search Specialists