Building a search engine with Clojure

Filip de Waard (fmw@vixu.com)

Code: https://github.com/fmw/alida

Why work on search?

- data is only useful if you can find it when you need it,
- fragmentation makes a lot of great data hard to work with,
- most applications that handle data benefit from having a search capability,
- generic search engines aren't good at specialized search,
- data mining & analytics to learn from the vast quantity of online information.

Why Clojure?

- the ability to access Java libraries from concise, beautiful Lisp code,
- functional programming and search are a good fit,
- no unnecessary concurrency headaches,
- be faster than the competition at implementing new features.

What does a search engine need to do?

- data retrieval (crawling),
- data storage (optional),
- data extraction (scraping),
- indexing,
- search,
- presentation.

Popular open source crawlers

Java:

- Apache Nutch: http://nutch.apache.org/
- Heritrix: https://github.com/internetarchive/heritrix3

Python:

- Scrapy: http://scrapy.org/
- Mechanize: http://wwwsearch.sourceforge.net/mechanize/

Other:

- wget: http://www.gnu.org/software/wget/
- HTTrack: http://www.httrack.com/page/1/en/index.html

Web crawling process

1. Download a seed page,
2. store page (optional),
3. extract links,
4. filter links,
5. download next page,
6. determine relevance,
7. repeat the steps for every relevant link.

Crawling approaches

Brute force:

- crawl and process everything you can get your hands on.

Directed crawling:

- specifically tell the crawler where to go.

Weighted crawling:

- start the crawler with a seed URI,
- run a scoring function over every new page,
- use the score to decide how to continue.

Roll your own crawler in Clojure

Leverage existing libraries:

- clj-http (wrapper for Apache HttpComponents),
- Enlive (selector-based data extraction & templating),
- Jsoup (Java HTML parser),
- Apache Lucene (search and indexing library written in Java).

Prototype:

- CouchDB through Clutch.

Recommended for production:

- Apache Hadoop for distributed crawling,
- Apache HBase and/or HDFS for storage.

Crawling considerations

Being polite:

- respect robots.txt,
- delay between requests.

Dealing with anti-crawling measures and non-standard content:

- headless browser for dynamic content (Selenium has good Clojure support),
- use OCR for images (e.g. tesseract-ocr),
- be aware of blocking through the user-agent header,
- don't get trapped in a loop,
- use a service like Amazon EC2 to recycle IP addresses.

Directed crawl

- Follow all links that start with "/en" on the seed page,
- only follow the external links in the content div on those pages,
- there is no :next key for that level, so the crawl stops there.

```
@(directed-crawl "external-links"      ;; crawl-tag
                 (util/make-timestamp) ;; crawl-timestamp
                 1000 ;; be nice and wait 1 second between requests
                 "http://www.vixu.com/"
                 [{:selector [:a]
                   :path-filter #"^/en.*"
                   :next [{:selector [[:div#content] [:a]]
                           :filter #"^(http://www.vixu.com/.*){0}.*"}]}]))
```

Directed crawl implementation

```
(defn directed-crawl
  [crawl-tag crawl-timestamp sleep-for seed-uri initial-selectors]
  (future
    (loop [todo-links [{:uri seed-uri :selectors initial-selectors}]
           crawled-links #{}
           results []]
      (if-let [{:keys [uri selectors]} (first todo-links)]
        (if (contains? crawled-links uri)
          (recur (rest todo-links) crawled-links results)
          (let [req (get-page uri)]
            (Thread/sleep sleep-for)
            (recur (concat (rest todo-links)
                           (get-crawlable-links-for-document uri
                                                             req
                                                             selectors))
                   (conj crawled-links uri)
                   (conj results (assoc req
                                        :type "crawled-page"
                                        :crawl-tag crawl-tag
                                        :crawl-timestamp crawl-timestamp
                                        :uri uri
                                        :crawled-at (util/make-timestamp))))))
        results))))
```

Weighted crawling scoring functions

Example page-scoring-fn that counts the number of occurrences of the string "website-management" on every requested page:

```
(defn page-scoring-fn [uri request]
  (count (re-seq #"website-management" (:body request))))
```

Example link-checker-fn that limits the crawl to Vixu.com:

```
(defn link-checker-fn [uri]
  (not (nil? (re-matches #"^http://www.vixu.com/.*" uri))))
```

Weighted crawl function

- recursively follows links,
- crawls external links in a new thread,
- saves the page to the database if the result from the page-scoring-fn is positive,
- doesn't follow links from a page when the page-scoring-fn returns a negative value,
- only follows links if calling link-checker-fn on them returns true.

```
@(weighted-crawl "alida"                    ;; database
                 "website-management-crawl" ;; crawl-tag
                 (util/make-timestamp)      ;; crawl-timestamp
                 1000                       ;; delay in ms
                 "http://www.vixu.com/"     ;; seed-uri
                 100                        ;; max-depth
                 page-scoring-fn
                 link-checker-fn)
```

Weighted crawl implementation

```
(defn weighted-crawl
  [database crawl-tag crawl-timestamp sleep-for seed-uri max-depth page-scoring-fn & [link-checker-fn]]
  (future
    (loop [todo-links [[seed-uri 0]]
           crawled-links #{}]
      (when-let [[active-uri depth] (first todo-links)]
        (if (and (not (contains? crawled-links active-uri))
                 (<= depth max-depth)
                 (or (not link-checker-fn) (link-checker-fn active-uri))
                 (not (crawled-in-last-hour? (db/get-page database crawl-tag active-uri))))
          (if (same-domain? active-uri seed-uri)
            (if-let [req (get-page active-uri)]
              (do
                (Thread/sleep sleep-for)
                (let [score (page-scoring-fn active-uri depth req)]
                  (when (pos? score)
                    (db/store-page database crawl-tag crawl-timestamp active-uri req score))
                  (recur (if (neg? score)
                           (rest todo-links)
                           (concat (rest todo-links)
                                   (map (fn [uri]
                                          [uri (inc depth)])
                                        (scrape/get-links-jsoup active-uri (:body req)))))
                         (conj crawled-links active-uri))))
              (recur (rest todo-links) (conj crawled-links active-uri)))
            (do
              (weighted-crawl database crawl-tag crawl-timestamp sleep-for active-uri max-depth page-scoring-fn link-checker-fn)
              (recur (rest todo-links) (conj crawled-links active-uri))))
          (recur (rest todo-links) (conj crawled-links active-uri)))))))
```

Data extraction

Tools:

- avoid regular expressions,
- Enlive is beautiful, idiomatic Clojure,
- Jsoup is a fast alternative written in Java,
- Selenium to access XMLHttpRequest-heavy pages.

Challenges:

- it's easy to extract data when you already know the structure of a page,
- things become really interesting when you don't (machine learning!)

Enlive examples

```
(defn get-links-enlive
  [current-uri s selector]
  (into #{}
        (map #(util/get-absolute-uri current-uri (:href (:attrs %)))
             (enlive/select (enlive/html-resource (StringReader. s)) selector))))

 (get-links-enlive "http://www.vixu.com/" my-html [:a])
 (get-links-enlive "http://planet.haskell.org/"
                   my-html
                   [[:.sidebar]
                    [:ul  (enlive/nth-of-type 1)]
                    [:li]
                    [:a (enlive/nth-of-type 1)]])
```

Processing crawl data

```
(scrape/extract-and-store-data
   "alida"                       ;; database
   "alida-test-crawl"            ;; crawl-tag
   "2012-05-22T22:51:35.297Z"    ;; crawl-timestamp
   (fn [raw-page]                ;; processing fn
      {:uri (:uri raw-page)
       :title (scrape/get-trimmed-content (:body raw-page) [:title])
       :fulltext (scrape/html-to-plaintext (:body raw-page))}))
```

Data extraction implementation

```
(defn extract-and-store-data
  [database crawl-tag crawl-timestamp scraping-fn & [batch-size]]
  (let [limit (or batch-size 1000)]
    (loop [raw (db/get-pages-for-crawl-tag-and-timestamp database
                                                         crawl-tag
                                                         crawl-timestamp
                                                         limit)]
      (db/add-batched-documents database
                                (map (fn [raw-page]
                                       (assoc (scraping-fn raw-page)
                                         :type "scrape-result"
                                         :uri (:uri raw-page)
                                         :crawled-at (:crawled-at raw-page)
                                         :crawl-tag crawl-tag
                                         :crawl-timestamp crawl-timestamp))
                                     (:documents raw)))
      (when-let [{:keys [uri timestamp id]} (:next raw)]
        (recur (db/get-pages-for-crawl-tag-and-timestamp database
                                                         crawl-tag
                                                         crawl-timestamp
                                                         limit
                                                         uri
                                                         timestamp
                                                         id))))))
```

Apache Lucene 4.0

- still unreleased, but already used in production by several big users,
- now using bytes (UTF-8) for indexing,
- flexible indexing to tune how data is stored,
- improved near-real-time (NRT) search,
- various performance improvements,
- improved concurrency,
- efficient fuzzy queries using Levenshtein Automaton,
- lots of breakage in backwards-compatibility.

Lucene Analyzers

Analyzers are used to tokenize strings. Picking the right analyzer depends on several factors, like the structure of your data and language. The StandardAnalyzer works quite well in most cases, however.

Wikipedia: "Tokenization is the process of breaking a stream of text up into words, phrases, symbols, or other meaningful elements called tokens."

```
(defn #^StandardAnalyzer create-analyzer
  []
  (StandardAnalyzer. (. Version LUCENE_40)))
```

Lucene Directories

Lucene has several types of index storage in the form of Directory subclasses, e.g.:

- RAMDirectory (for small indexes),
- MMapDirectory (hybrid between memory and disk storage),
- SimpleFSDirectory (a simple filesystem directory),
- NIOFSDirectory (directory with positional read; broken on Windows),
- CompoundFileDirectory (read-only directly stored in a single file).

Alida uses RAMDirectory for tests and NIOFSDirectory as an option for real indexes, but use MMapDirectory on high-memory production machines.

```
(defn create-directory
  [path]
  (if (= path :RAM)
    (RAMDirectory.)
    (NIOFSDirectory. (File. path))))
```

Lucene IndexReader implementations

Lucene has several ways of reading indexes. IndexReader is now read-only in 4.0. There are several options to manage IndexReader instances. These manager classes are useful for near-real-time search and making sure users get a consistent index during pagination, instead of a potentially reordered one.

```
(defn #^IndexReader create-index-reader
  [#^Directory directory]
    (try
      (. IndexReader open directory)
      (catch IndexNotFoundException e
        nil)))
```

Lucene IndexWriter

Lucene IndexWriter objects can be tracked by near-real-time search managers, but here is a simple example:

```
(defn #^IndexWriter create-index-writer
  [analyzer directory mode]
  (let [config (IndexWriterConfig. (Version/LUCENE_40) analyzer)]
    (doto config
      (.setRAMBufferSizeMB 49)
      (.setOpenMode (cond
                     (= mode :create)
                     (IndexWriterConfig$OpenMode/CREATE)
                     (= mode :append)
                     (IndexWriterConfig$OpenMode/APPEND)
                     (= mode :create-or-append)
                     (IndexWriterConfig$OpenMode/CREATE_OR_APPEND))))
      (IndexWriter. directory config)))

(defn add-documents-to-index!
  [writer fields-map document-value-maps]
  (doseq [doc document-value-maps]
    (.addDocument writer (create-document- fields-map doc))))

(with-open [writer (create-index-writer my-analyzer my-directory :create)]
  (add-documents-to-index! writer
                           {:fulltext (create-field "fulltext" "" :indexed :tokenized)}
                           [{:fulltext "Hic sunt dracones."}]))
```

Lucene Documents

Documents are a collection of fields:

```
(defn #^Document create-document-
  [fields-map values-map]
  (let [doc (Document.)]
    (doseq [[k field] fields-map]
      (.add doc (set-field-value! field (get values-map k))))
    doc))
```

Can be converted back to Clojure maps (limited to stored fields when reading from an index):

```
(defn document-to-map
  [doc]
  (apply merge
         (map (fn [field]
                {(keyword (.name field))
                 (or (.numericValue field) (.stringValue field))})
              (.getFields doc))))
```

The new FieldType class

Field is now separate from FieldType:

```
(defn #^FieldType create-field-type [data-type & options]
  (let [field-type (FieldType.)]
    (cond
     (= data-type :double)
     (.setNumericType field-type FieldType$NumericType/DOUBLE)
     (= data-type :float)
     (.setNumericType field-type FieldType$NumericType/FLOAT)
     (= data-type :int)
     (.setNumericType field-type FieldType$NumericType/INT)
     (= data-type :long)
     (.setNumericType field-type FieldType$NumericType/LONG))

    (doto field-type
      (.setIndexed (not (nil? (some #{:indexed} options))))
      (.setStored (not (nil? (some #{:stored} options))))
      (.setTokenized (not (nil? (some #{:tokenized} options))))
      (.freeze))))
```

Creating Field objects

Polymorphism can be a good thing; not blasphemy that should be avoided in functional programming!

```
(defmulti #^Field create-field
  (fn [name value & options]
    (class value)))

(defmethod create-field java.lang.String [name value & options]
  (Field. name (or value "") (apply (partial create-field-type :string) options)))

(defmethod create-field java.lang.Long [name value & options]
  (LongField. name (or value 0) (apply (partial create-field-type :long) options)))

(defmethod create-field java.lang.Integer [name value & options]
  (IntField. name (or value (int 0)) (apply (partial create-field-type :int)  options)))

(defmethod create-field java.lang.Float [name value & options]
  (FloatField. name (or value (float 0.0)) (apply (partial create-field-type :float) options)))

(defmethod create-field java.lang.Double [name value & options]
  (DoubleField. name (or value 0.0) (apply (partial create-field-type :double) options)))

(defn #^LongField create-date-field
  [name rfc3339-date-string & options]
  (apply (partial create-field name (util/rfc3339-to-long rfc3339-date-string)) options))
```

Mutating existing Field objects

The Lucene documentation recommends mutating existing Field objects over creating them from scratch for every Document for performance reasons. This is less kosher in terms of functional programming, but still the pragmatic option. It means that documents need to be written immediately, however, otherwise the Field values could change!

```
(defmulti #^Field set-field-value!
  (fn [field value]
    (class value)))

(defmethod set-field-value! java.lang.String [field value]
  (doto field
    (.setStringValue value)))

(defmethod set-field-value! java.lang.Long [field value]
  (doto field
    (.setLongValue value)))

(defmethod set-field-value! java.lang.Integer [field value]
  (doto field
    (.setIntValue value)))

(defmethod set-field-value! java.lang.Float [field value]
  (doto field
    (.setFloatValue value)))

(defmethod set-field-value! java.lang.Double [field value]
  (doto field
    (.setDoubleValue value)))
```

Lucene TermQuery

Lucene implements a plethora of query types. The most basic and ubiquitous one is probably the TermQuery, which searches documents for the provided value in a given field:

```
(defn #^TermQuery create-term-query
  [field value]
  (TermQuery. (Term. field value)))
```

Lucene NumericRangeQuery

NumericRangeQuery instances are an efficient way to look for numbers in a specified range.

```
(defmulti #^NumericRangeQuery create-numeric-range-query
  (fn [field-name min max]
    (class min)))

(defmethod create-numeric-range-query java.lang.Long [field-name min max]
  (when (and (= (class min) java.lang.Long) (= (class max) java.lang.Long) (>= max min))
    (NumericRangeQuery/newLongRange field-name min max true true)))

(defmethod create-numeric-range-query java.lang.Float [field-name min max]
  (when (and (= (class min) java.lang.Float) (= (class max) java.lang.Float) (>= max min))
    (NumericRangeQuery/newFloatRange field-name min max true true)))

(defmethod create-numeric-range-query java.lang.Double [field-name min max]
  (when (and (= (class min) java.lang.Double) (= (class max) java.lang.Double) (>= max min))
    (NumericRangeQuery/newDoubleRange field-name min max true true)))

(defmethod create-numeric-range-query java.lang.Integer [field-name min max]
  (when (and (= (class min) java.lang.Integer) (= (class max) java.lang.Integer) (>= max min))
    (NumericRangeQuery/newIntRange field-name min max true true)))

(defn #^NumericRangeQuery create-date-range-query
  [field-name start-date-rfc3339 end-date-rfc3339]
  (create-numeric-range-query field-name
                              (util/rfc3339-to-long start-date-rfc3339)
                              (util/rfc3339-to-long end-date-rfc3339)))
```

Chaining Query instances with a BooleanQuery

```
(defn #^BooleanQuery create-boolean-query
  [& pairs]
  (when (even? (count pairs))
    (let [bq (BooleanQuery.)]
      (doseq [[query clause] (partition 2 pairs)]
        (.add bq
              query
              (cond
                (= clause :must) BooleanClause$Occur/MUST
                (= clause :must-not) BooleanClause$Occur/MUST_NOT
                (= clause :should) BooleanClause$Occur/SHOULD)))
      (when (pos? (alength (.getClauses bq)))
        bq))))

 (create-boolean-query my-term-query :must
                       my-numeric-range-query :must)
```

Lucene filters

Filters are an addendum to a search query, that fine-tune the results of the search.

A very convenient way to use filters is to create regular queries and convert those into a Filter using QueryWrapperFilter:

```
(defn #^QueryWrapperFilter create-query-wrapper-filter
  [query]
  (when query
    (QueryWrapperFilter. query)))
```

Lucene search

```
(defn search
  [query filter fulltext-field limit reader analyzer & [after-doc-id after-score]]
  (if (and reader (not-empty query))
    (try
      (let [searcher (IndexSearcher. reader)
            q (.parse (StandardQueryParser. analyzer) query fulltext-field)
            top-docs (if (nil? after-doc-id)
                       (if (nil? filter)
                         (.search searcher q limit)
                         (.search searcher q filter limit))
                       (if (nil? filter)
                         (.searchAfter searcher (ScoreDoc. after-doc-id after-score) q limit)
                         (.searchAfter searcher (ScoreDoc. after-doc-id after-score) q filter limit)))]
        {:total-hits (.totalHits top-docs)
         :docs (map (fn [score-doc doc]
                      (assoc (document-to-map doc)
                        :index {:doc-id (.doc score-doc)
                                :score (.score score-doc)}))
                    (.scoreDocs top-docs)
                    (get-docs reader (.scoreDocs top-docs)))})
      (catch ParseException e nil)
      (catch TokenMgrError e nil)
      (catch QueryNodeParseException e nil))
    {:total-hits 0
     :docs nil}))
```