Indexing and Search with Lucene @Greplin

About Greplin

+ More!

The Nature of our Service

- Volume of insertions >>> Volume of searches
  - Peak insertion rate has peaked to 5k documents / second
  - Fully loaded search server is responsible for up to 80M docs averaging 3kB each
- Low per-user cost
  - Each machine is currently a Amazon c1.medium with 1.7 GB of RAM
  - Currently handles about 50M documents per GB of RAM
- Smart write buffering, smart index sharding, and smart cache warming are critical to our service
  - Users expect near-real-time search for their real-time data

Some Cool Hacks

- To help keep search latency down, we defer, batch, and reduce disk operations
  - Index "writes" are actually insertions to a RAMDirectory and an entry in a write-ahead log
  - During a "flush" we actually update users' separate on-disk indexes
  - Use a highly efficient Bloom Filter to reduce the number of updates by eliminating duplicates
    - (It's on Github!)
- Also, caching
  - Cache warming after logins, Chrome extension focus

Other Cool Hacks

- Customized spelling suggestions for misspelled user queries
  - Need separate per-index spelling indices
    - Privacy
    - Performance
    - Quality
- Problem?
  - Lots of duplicated content between spelling indices
  - User-misspelled words entering their own spelling index

Other Cool Hacks

- Solution: fork the SpellChecker class, build a two-tiered spellchecking system
  1. Build a global spelling index based on dictionary words
  2. User indices filter out words present in the global spelling index
  3. User indices filter out words which occur under a certain frequency
- Result: decreased the size of English per-user spelling indices by ~80%

Interesting Tokenization Problems

Interesting Tokenization Problems

- Typical things like underscores and dashes
  - We index Dropbox as well as email, so imagine a query for {business_cat} matching an email containing
    - "A business manager named Catherine"
      - vs
    - business_cat.jpg
  - Solution: turn tokens connected by underscores and dashes into phrase queries

- Less typical things such as #hashtags and @handles vs. addresses (Suite #106) and email addresses

Interesting Tokenization Problems

- Solution: Keep meaningful punctuation and discard the boring kinds
  - Index {#} ,{hashtag}, {@}, {handle} separately
  - Treat queries of this nature as phrase queries
  - Queries for {1300 Island Drive #106} and {1300 Island Drive 106} will both have exact term matches for the address

- Problem?
  - The StandardTokenizer doesn't really play nice here

Ultimately...

- We forked the StandardTokenizer JFlex and built our own lexer
  - Which really wasn't so bad, in all honesty
- Unfortunately, query parsing still requires some hacks
  - Our QueryParser gets "business", "cat" from the Tokenizer
  - We need to examine the token breaks in the original user query
    - Prime example: my_email@michaelcvet.com vs @michaelcvet

Related:

- We often run into interesting problems such as these and want to share our solutions!
  - github.com/Greplin
    - All kinds of stuff
  - github.com/Greplin/lucene-interval-fields
    - Useful utilities for range-based queries and analysis
  - github.com/Greplin/greplin-lucene-utils
    - A collection of Collectors and Query classes
  - github.com/Greplin/greplin-bloom-filter
    - An efficient Bloom Filter implementation, in Java

Questions?
Comments?
Suggestions?

Mike Cvet

mike@greplin.com

@michaelcvet