Finite-State Queries in Lucene

Robert Muir
rmuir@apache.org

Agenda

- Introduction to Lucene
- Improving inexact matching:
  - Background
  - Regular Expression, Wildcard, Fuzzy Queries
- Additional use cases:
  - Language support: expansion versus stemming
  - Improved spellchecking

Introduction to Lucene

- Open Source Search Engine Library
  - Not just Java, ported to other languages too.
  - Commercial support via several companies
- Just the library
  - Embed for your own uses, e.g. Eclipse
  - For a search server, see Solr
  - For web search + crawler, see Nutch
- Website: http://lucene.apache.org

Exact queries

- Typical searches are exact match.
- Terms normalized the same at index/query
  - Index time: Dogs -> dog
  - Query time: dog -> dog
- Fast
  - Analyzers normalize documents terms and query terms.
  - Searching is exact match on normalized terms.

Inexact queries

- Exact works well for many use cases
  - Stemming for full-text document search
- But other use cases demand more:
  - Pattern search: Regexp, Wildcard, Glob, …
  - Fuzzy search: Levenshtein, …
  - Range search: Numeric ranges
- Traditionally slow and not very scalable
  - Often involved inspecting **every** indexed term

How to improve?

- Improve the computational complexity
- Two components: number of terms examined, and comparison complexity.
- Example worst case: FuzzyQuery
  - O(t) terms examined, t=number of terms in all docs for that field. Exhaustively compares each term. We would prefer $O(\log_2 t)$ instead.
  - $O(n^2)$ comparison function, n=length of term. Levenshtein dynamic programming. We would prefer O(n) instead.

Complexity: # of terms

- Imagine terms dictionary as a tree
- Before 2.9, queries only operate on one "subtree".
  - For example, Regex and Fuzzy exhaustively evaluate all terms, unless you give them a "constant prefix".
- It's a tree-like structure*, lets make use of this to get $O(\log_2 t)$ inspections if we can!
- http://lucene.apache.org/java/3_0_1/fileformats.html

Complexity: comparisons

- The matching function itself can be important, even if we inspect less terms.
- Comparison of just the matching function*:

Engine	10k benchmark iterations (ms)
Jakarta	4,594ms
JDK	609ms
Brics Automaton	172ms

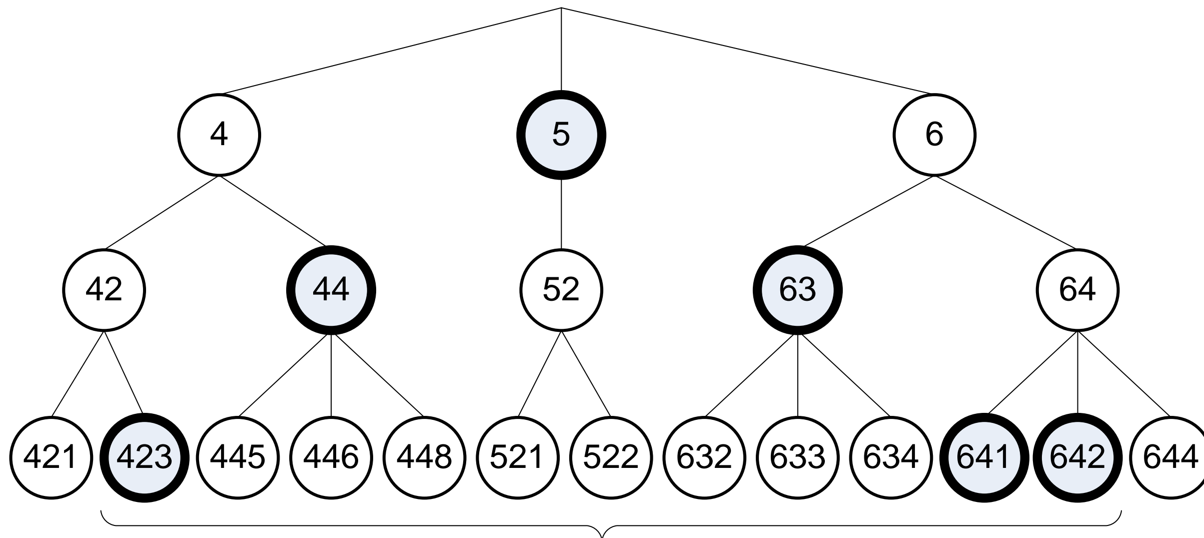
- Automaton matching: O(n) where n=word length
  - Independent of pattern complexity, transition function:

```
return transitions[state * points.length + classmap[c]];
```

- http://tusker.org/regex/regex_benchmark.html

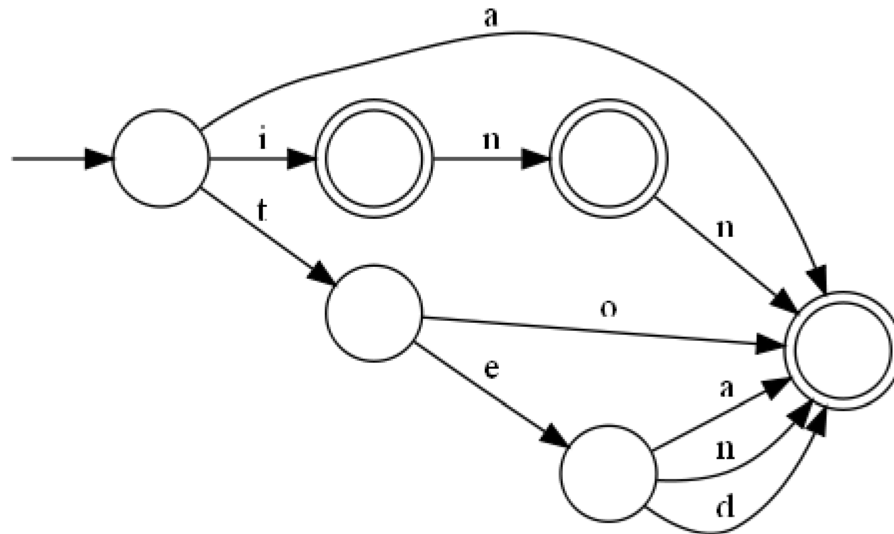
Lucene 2.9: Fast Numeric Ranges

- Indexes at different levels of precision.
- Enumerates multiple subtrees.
  - But typically this is a small number: e.g. 15
- Query APIs improved to support this.



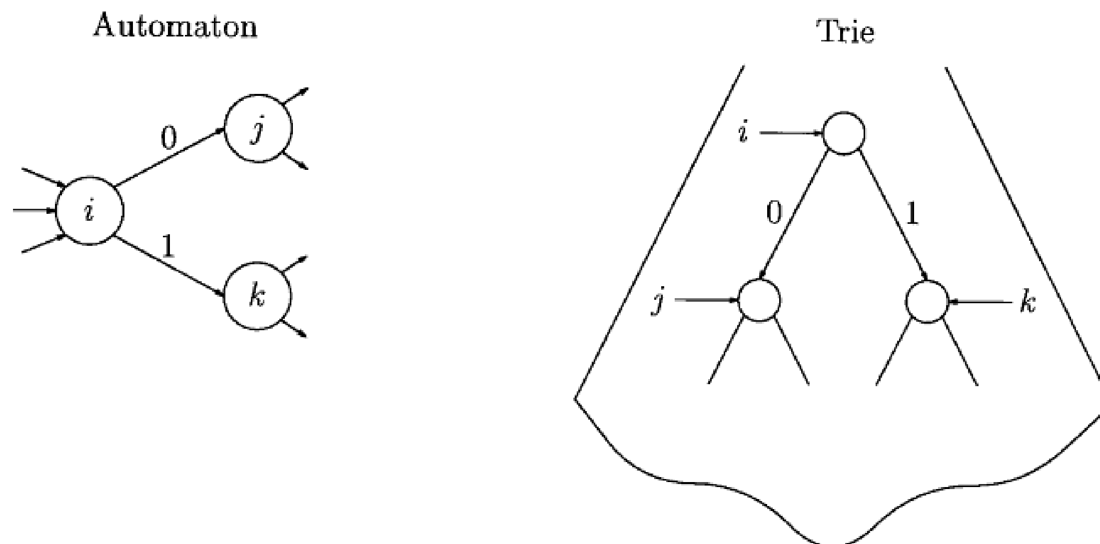
Automaton Queries

- Only explore subtrees that can lead to an accept state of some finite state machine.
- AutomatonQuery traverses the term dictionary and the state machine in parallel



Another way to think of it

- Index as a state machine that recognizes Terms and transduces matching Documents.
- AutomatonQuery represents a user's search need as a FSM.
- The intersection of the two emits search results.

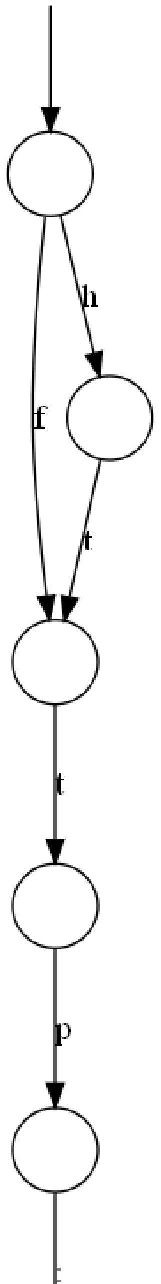


Query API improvements

- Automata might need to do many seeks around the term dictionary.
  - Depends on what is in term dictionary
  - Depends on state machine structure
- MultiTermQuery API further improved
  - Easier and more efficient to skip around.
  - Explicitly supports seeking.

Regex, Wildcard, Fuzzy

- Without constant prefix, exhaustive
  - Regex: (http|ftp)://foo.com
  - Wildcard: ?oo?ar
  - Fuzzy: foobar~
- Re-implemented as automata queries
  - Just parsers that produce a DFA
  - Improved performance and scalability
  - (http|ftp)://foo.com examines 2 terms.



Roll your own: FuzzyPrefixQuery

```
// a term representative of the query, containing the field.
// term text is not important and only used for toString() and such
Term term = new Term("yourfield", "bla~*");

// builds a DFA for all strings within an edit distance of 2 from "bla"
Automaton fuzzy = new LevenshteinAutomata("bla").toAutomaton(2);

// concatenate this with another DFA equivalent to the "*" operator
Automaton fuzzyPrefix = BasicOperations.concatenate(fuzzy,
        BasicAutomata.makeAnyString());

// build a query, search with it to get results.
AutomatonQuery query = new AutomatonQuery(term, fuzzyPrefix);
```

Additional/Advanced Use Cases

Stemming

- Stemmers work at index and query time
  - walked, walking -> walk
  - Can increase retrieval effectiveness
- Some problems
  - Mistakes: international -> intern
  - Must determine language of documents
  - Multilingual cases can get messy
  - Tuning is difficult: must re-index
  - Unfriendly: wildcards on stemmed terms…

Expansion instead

- Stemming is just a form of query expansion.
- Don’t remove data at index time
  - Expand the query instead.
  - Single field now works well for all queries: exact match, wildcard, expanded, etc.
- Simplifies search configuration
  - Tuning relevance is easier, no re-indexing.
  - No need to worry about language ID for docs.
  - Multilingual case is much simpler.

Automata expansion

- Natural fit for morphology
- Use set intersection operators
  - Minus to subtract exact match case
  - Union to search multiple languages
- Efficient operation
  - Doesn't explode for languages with complex morphology

Experimental results

- 125k docs English test collection
  - Results are for TD queries
- Inverted the “S-Stemmer”
  - 6 declarative rewrite rules to regex
- Competitive with traditional stemming.

	No Stemming	Porter	S-Stem	Automaton S-Stem
MAP	0.4575	0.5069	0.5029	0.4979
MRR	0.8070	0.7862	0.7587	0.8220
# Terms	336,675	280,061	305,710	336,675

TODO:

- Support expansion models, too in Lucene.
- Language-specific resources
  - lucene-hunspell could provide these
- Language-independent tokenization
  - Unicode rules go a long way.
- Scoring that doesn't need stopwords
  - For now, use CommonGrams!

spellcheckking

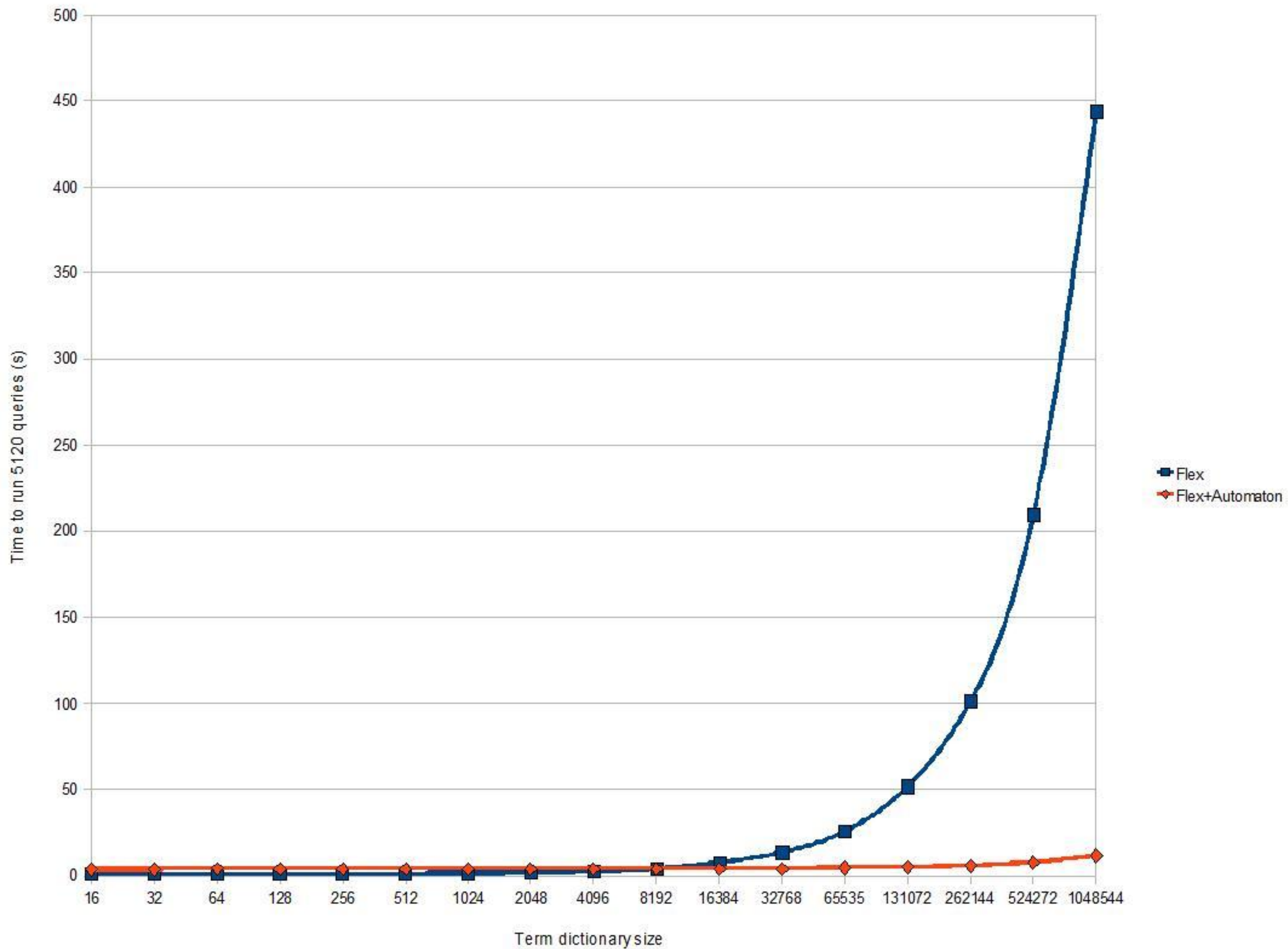
- Lucene spellchecker builds a separate index to find correction candidates
- Perhaps our fuzzy enumeration is now fast enough for small edit distances (e.g. 1,2) to just use the index directly.
- Could simplify configurations, especially distributed ones.

Conclusions

Conclusions:

- In an upcoming version of Lucene, you can achieve much more scalable inexact matching.
- Less comparisons, faster comparisons.
- Full Unicode support: no cheating.
- For advanced uses, you can write your own queries as finite-state queries.

Backup slides



Time to run 5120 queries (s)
500
450
400
350
300
250
200
150
100
50
0
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65535
131072
262144
524272
1048544
Term dictionary size
Flex
Flex+Automaton