# Lucene 4 - Revisiting problems for speed

Simon Willnauer
Lucene Core-Committer & PMC Member
simonw@apache.org

# About me

- Lucene - Core Committer & PMC Member
- Freelancer on Lucene & Solr
- Co-Organizer of BerlinBuzzwords

# What is this all about?

- Most of the talks show improvements and what they do
- This talk is an experiment on showing how we improved it
- The following slides will:
  - introduce you to problems that existed for years in Lucene <= Version 3.x
  - show their problem domain
  - and the key parts of the solution which might include some lines of code too

# Postinglist in memory

Lots of object for lots of terms

# Inverted Index?

| | |
|---|---|
| 1 | The old night keeper keeps the keep in the town |
| 2 | In the big old house in the big old gown. |
| 3 | The house in the town had the big old keep |
| 4 | Where the old night keeper never did sleep. |
| 5 | The night keeper keeps the keep in the night |
| 6 | And keeps in the dark and sleeps in the light. |

Table with 6 documents

Example from:
*Justin Zobel , Alistair Moffat,*
*Inverted files for text search engines,*
*ACM Computing Surveys (CSUR)*
*v.38 n.2, p.6-es, 2006*

# Inverted Index?

| | |
|---|---|
| 1 | The old night keeper keeps the keep in the town |
| 2 | In the big old house in the big old gown. |
| 3 | The house in the town had the big old keep |
| 4 | Where the old night keeper never did sleep. |
| 5 | The night keeper keeps the keep in the night |
| 6 | And keeps in the dark and sleeps in the light. |

Table with 6 documents

<< index >>

? Lucene

| term | freq | Posting list |
|---|---|---|
| and | 1 | 6 |
| big | 2 | 2 3 |
| dark | 1 | 6 |
| did | 1 | 4 |
| gown | 1 | 2 |
| had | 1 | 3 |
| house | 2 | 2 3 |
| in | 5 | 1 2 3 5 6 |
| keep | 3 | 1 3 5 |
| keeper | 3 | 1 4 5 |
| keeps | 3 | 1 5 6 |
| light | 1 | 6 |
| never | 1 | 4 |
| night | 3 | 1 4 5 |
| old | 4 | 1 2 3 4 |
| sleep | 1 | 4 |
| sleeps | 1 | 6 |
| the | 6 | 1 2 3 5 6 |
| town | 2 | 1 3 |
| where | 1 | 4 |

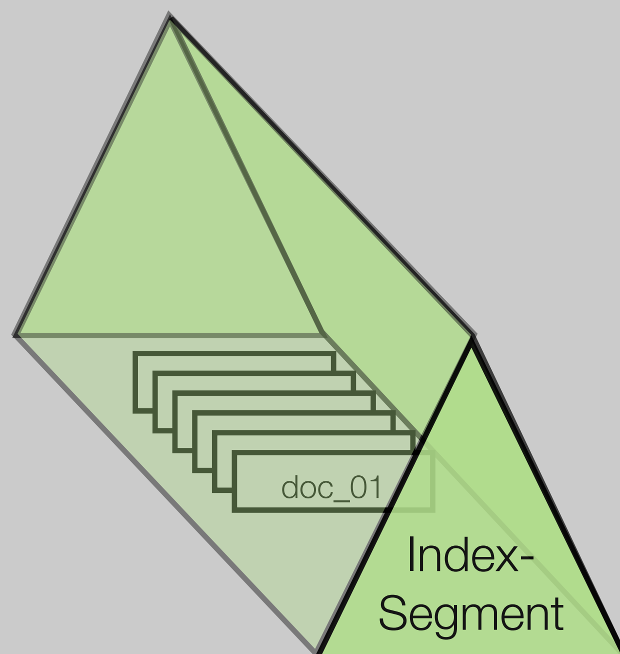
# During indexing....

- Lucene builds the TermDictionary & Postings in RAM until they get flushed to disc

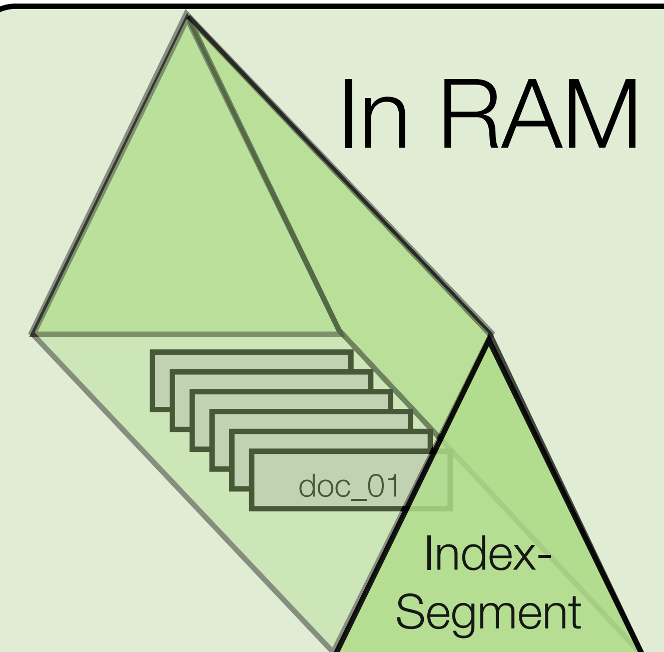
IndexReader / Searcher

IndexWriter

On Disk




In RAM

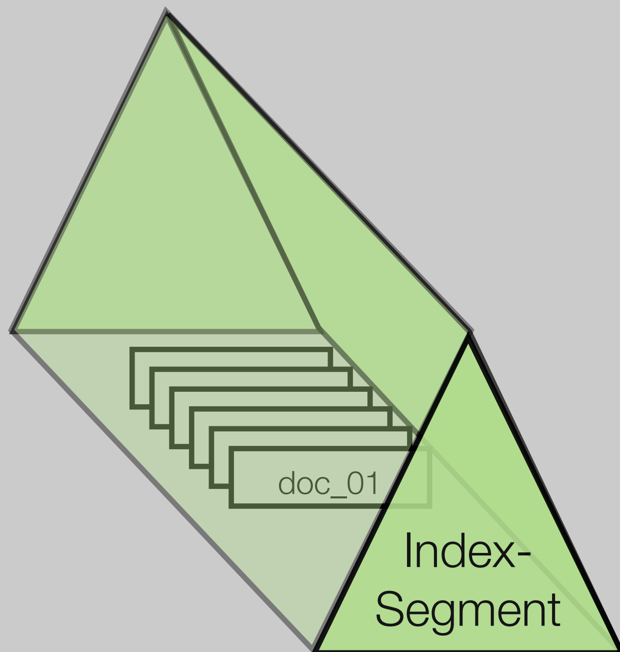


# During indexing....

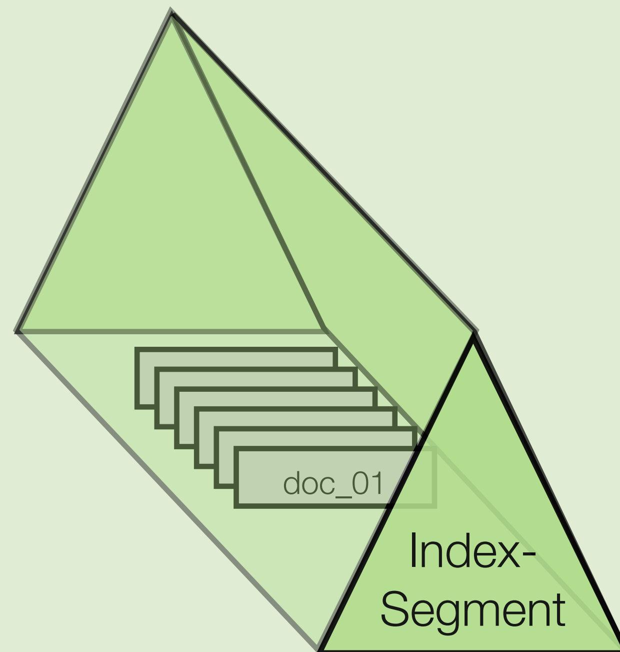
IndexReader / Searcher

IndexWriter

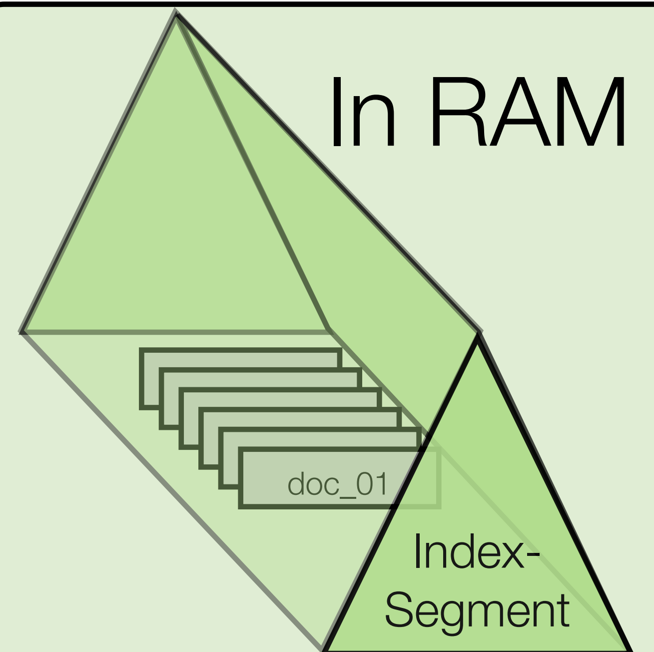
On Disk




On Disk




In RAM




Flush

# Until Lucene 3.0...

- PostingLists where represented by an array of `PostingList` - a struct-like class

```
class PostingList {
   int textPointer;
   int postingsPointer;
   int frequency;
}
```

- For each unique term in the dictionary a new instance of this class was created and added to the `PostingsList[ ]`

- Object creation in Java is fast - yes! Arrays are efficient - yes! Garbage Collection is slow - yes! :)

# Lucene 4 switched to parallel arrays instead....

- To overcome the need of doing too many garbage collections but still make use of Java's fast memory allocation PostingList was moved from Array-of-Objects to Object-of-Arrays.

| class PostingList { | class ParallelPostingsArray { |
| --- | --- |
| int textPointer; | int[ ] text; |
| int postingsPointer; | int[ ] postings; |
| int frequency; | int[ ] frequencies; |
| } | } |

- This reduced the number of long living objects dramatically

- Improved indexing performance dramatically when memory is tight

# Realtime Search - already committed improvement

## ParallelPostingsArray - LUCENE-2329

```
class PostingList {
   int textPointer;
   int postingsPointer;
   int frequency;
}
```

```
class ParallelPostingsArray {
  int[] text;
  int[] postings;
  int[] frequencies;
}
```

- Instead of PostingList[ ] use ParallelPostingsArray
- Reduces the number of long living objects dramatically
- Dramatic speed improvements when memory is tight (up to 400% according to buschmi@apache.org)

# Improvements due to Parallel Arrays

- Total number of objects is now constant and independent of the number of unique terms
- Parallel Arrays safe 28 bytes per unique term -> 41 % savings compared to Object-of-Array model
- Indexing 1 Million wikipedia document
  - -Xmx2G MaxRamBufferSizeMD = 200 -- 4.3% improvement
  - -Xmx256M MaxRamBufferSizeMD = 200 -- 86.5% improvement
- Important once RAM - Searchable buffers are used which utilize lots of memory

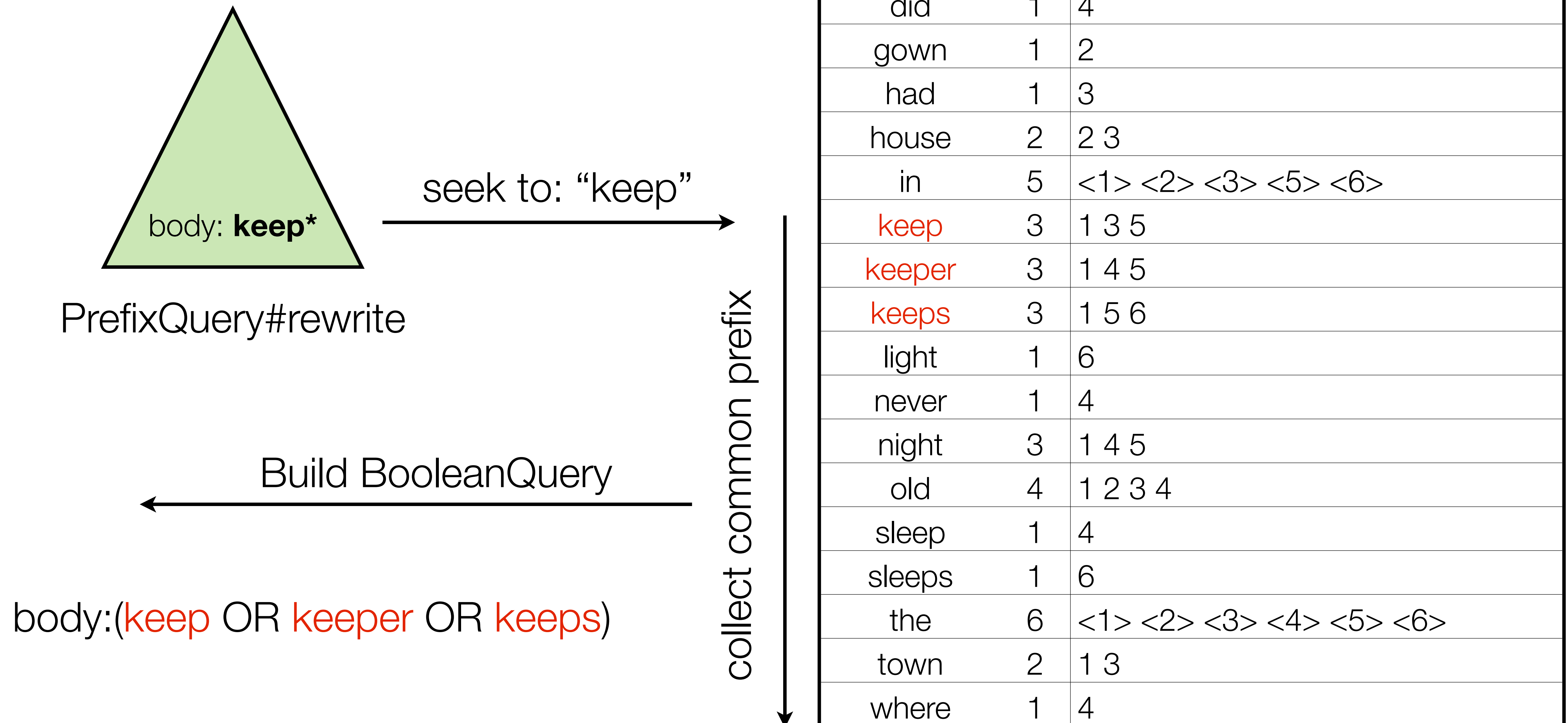
# MultiTermQuery

Traditionally very Slow!

# MultiTermQueries - traditionally very slow

- What is a MultiTermQuery in Lucene
  - a query that “rewrites” to more than a single term
  - for instance **PrefixQuery(“foo*”)** matches all documents starting with foo
- Lucene internally doesn’t have a specialized data structure for each high level query.
- MultiTermQuery instead creates a Boolean OR query during the “Rewrite-Process”

# Closer Look into Query#rewrite




| term | freq | Posting list |
|---|---|---|
| and | 1 | 6 |
| big | 2 | 2 3 |
| dark | 1 | 6 |
| did | 1 | 4 |
| gown | 1 | 2 |
| had | 1 | 3 |
| house | 2 | 2 3 |
| in | 5 | <1> <2> <3> <5> <6> |
| keep | 3 | 1 3 5 |
| keeper | 3 | 1 4 5 |
| keeps | 3 | 1 5 6 |
| light | 1 | 6 |
| never | 1 | 4 |
| night | 3 | 1 4 5 |
| old | 4 | 1 2 3 4 |
| sleep | 1 | 4 |
| sleeps | 1 | 6 |
| the | 6 | <1> <2> <3> <4> <5> <6> |
| town | 2 | 1 3 |
| where | 1 | 4 |

# What about more complex MTQ?

- PrefixQuery performance is OK compared to other MTQ
  - FuzzyQuery for instance  does less than 0.5 QPS (Queries per Second) on a fast machine if number of unique terms is high. (12 Cores 12GB RAM)
  - WildcardQuery is insane slow
- Fuzzy query for instance needs to examine (almost) every term in the term dictionary and calculate its Levenshtein distance during rewrite if no constant prefix is used.

# How to improve?

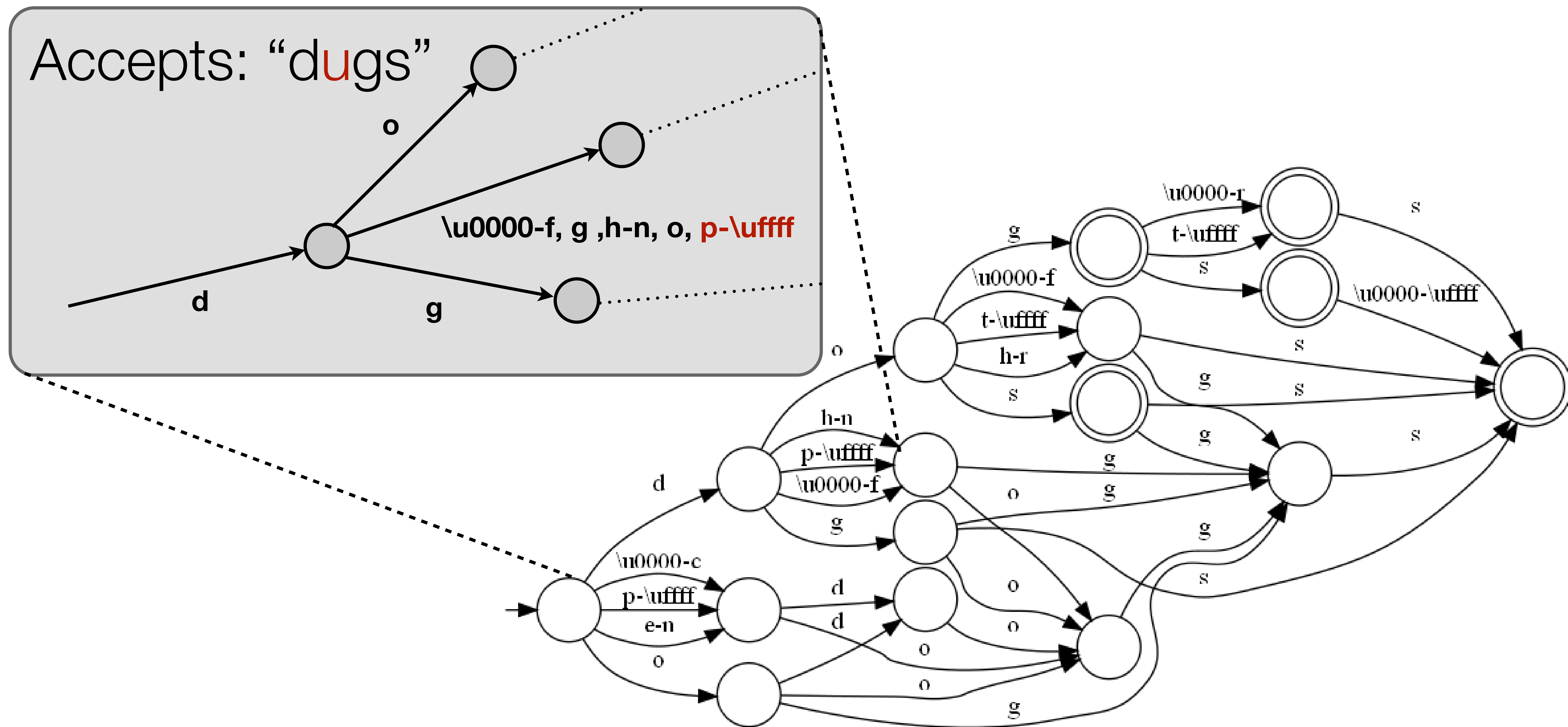
- Improve the computational complexity
- Two components: number of terms examined, and comparison complexity.
- Example worst case: FuzzyQuery
  - $O(t)$ terms examined, t=number of terms in all docs for that field. Exhaustively compares each term. We would prefer $O(\log_2 t)$ instead.
  - $O(n^2)$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

# Automaton Query

- 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
- Imagine the index as a state machine that recognizes Terms (UTF-8 bytes) and consumes matching Documents.
  - AutomatonQuery represents a user's search needs as a FSM.
  - The intersection of the two emits search results

# Automaton Query - Example FuzzyQuery

## Example FSM for the term “dogs~1”



# Automaton Query - Benchmark Results

| Query | QPS (3.x) | QPS (4.0) | Pct diff |
|---|---|---|---|
| united~0.6 | 0.41 | 24.70 | 5858.2% |
| united~0.7 | 0.44 | 94.77 | 21454.8% |

Optimized 7M Document Wikipedia index
Dual 6 Core Xeon / 12GB RAM

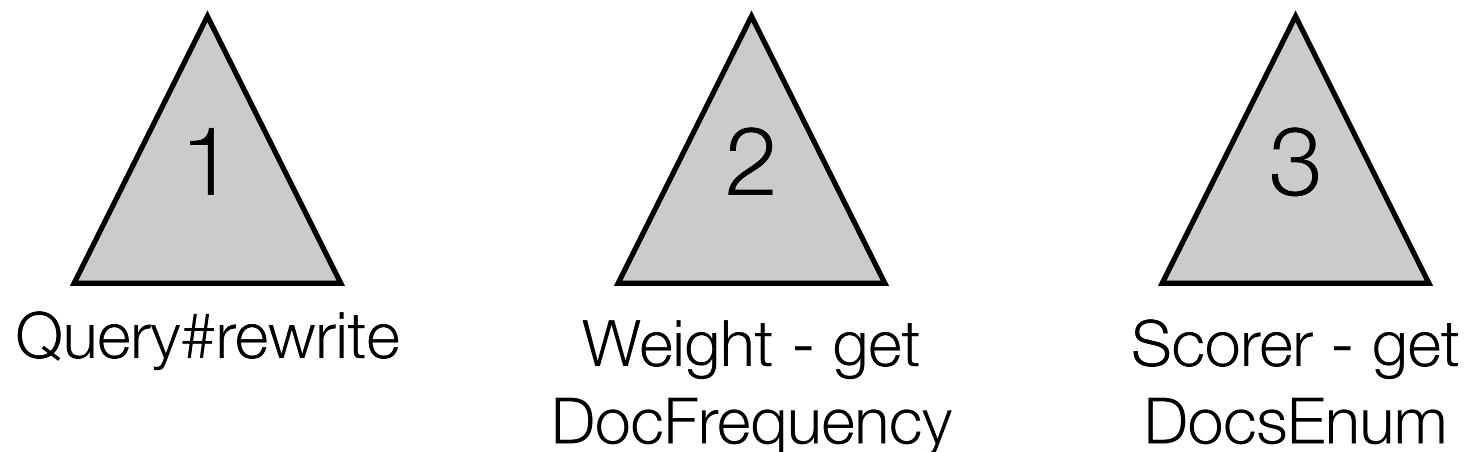
# More on MultiTermQuery

Making MTQ single pass

# More on MultiTermQuery...

- until 2 weeks ago every MultiTermQuery hits the TermDictionary 3x per term until the query gets scored

- On an multi-segment index this gets even worse



# Each time...

- Each time we hit the TermDictionary the Term needs to be looked-up
- Term-Lookups are fast...
  - TermDictionary has a LRU Cache & data-structures are optimized
- But still lots of unnecessary low level access:
  - MTQ rewrite to hundreds or thousands of terms
  - Pollute LRU Cache - slow down query performance under load
  - Caches don’t cache missed terms

# Solution - lookup terms in O(1)

- Since Rewrite & Scoring is guaranteed to happen on the same low-level index Term-Lookups can be done in constant time
  - Term Dictionary now allows to pull its internal state
  - Class `TermState` holds internal file pointers to reposition the Dictionary in constant time.
- Term-Lookups in segments not containing the term are caught early
- DocumentFrequency is encapsulated in `TermState`

# Improvement

- Since the term dictionary needs to do costly operations only once per term & segment MTQ scale much better
- LRU Cache is touched anymore since term lookup is O(1)
- Even under low-load situations this brings nice speedups

| Query | QPS base | QPS termstate | Pct diff |
|---|---|---|---|
| unit~2.0 | 10.04 | 10.59 | 5.4% |
| united~1.0 | 16.84 | 18.13 | 7.7% |
| unit~1.0 | 10.09 | 10.99 | 8.9% |
| un*d | 11.96 | 21.63 | 80.8% |
| unit* | 7.60 | 14.23 | 87.3% |
| u*d | 2.22 | 4.17 | 87.8% |
| uni* | 1.83 | 3.53 | 93.7% |

# Thank you for your attention