文本和图像搜索

DM-Team
2018-11

提纲

- 文本搜索-Elastic Search
  - 索引文档
    - 倒排
    - FST 有限状态转换器
    - Skip List 跳表
  - 查询文档
    - 提高Recall
    - 提高Precision
    - 相关度计算
    - 性能分析

提纲

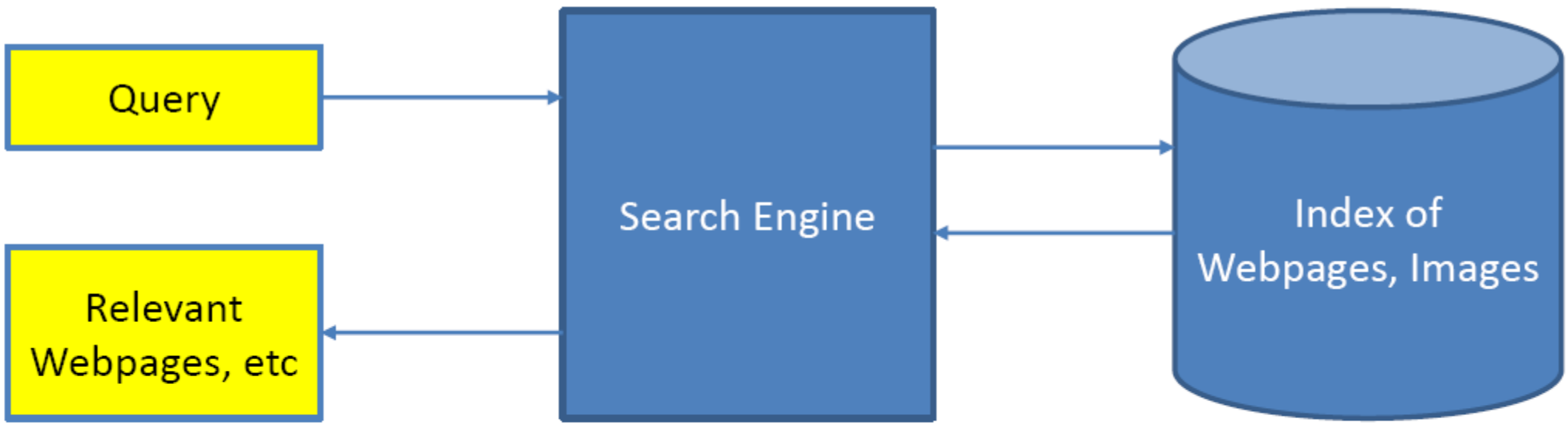
- 向量搜索-Faiss
  - NN,KNN,ANN
  - Quantization 量化
  - Product Quantization 乘积量化
  - Benchmark

Overview of Search Engine

Information pull: a user pulls information by making a specific request

User intent is explicitly reflected in query:
- Keywords, questions

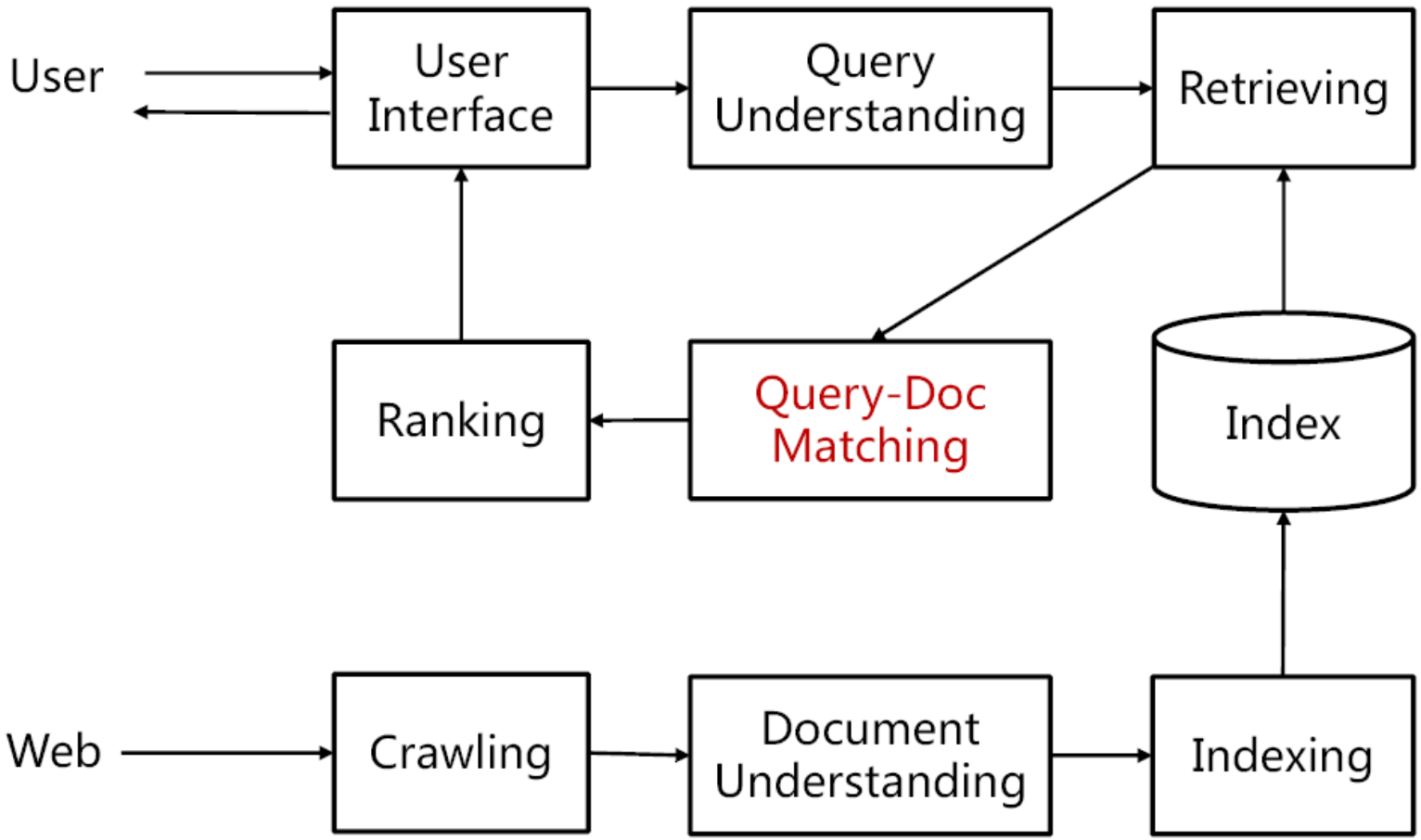
Content is in
- Webpages, images, …



Key challenge: query-document semantic gap

Elastic Search-索引文档

Overview of Web Search Engine



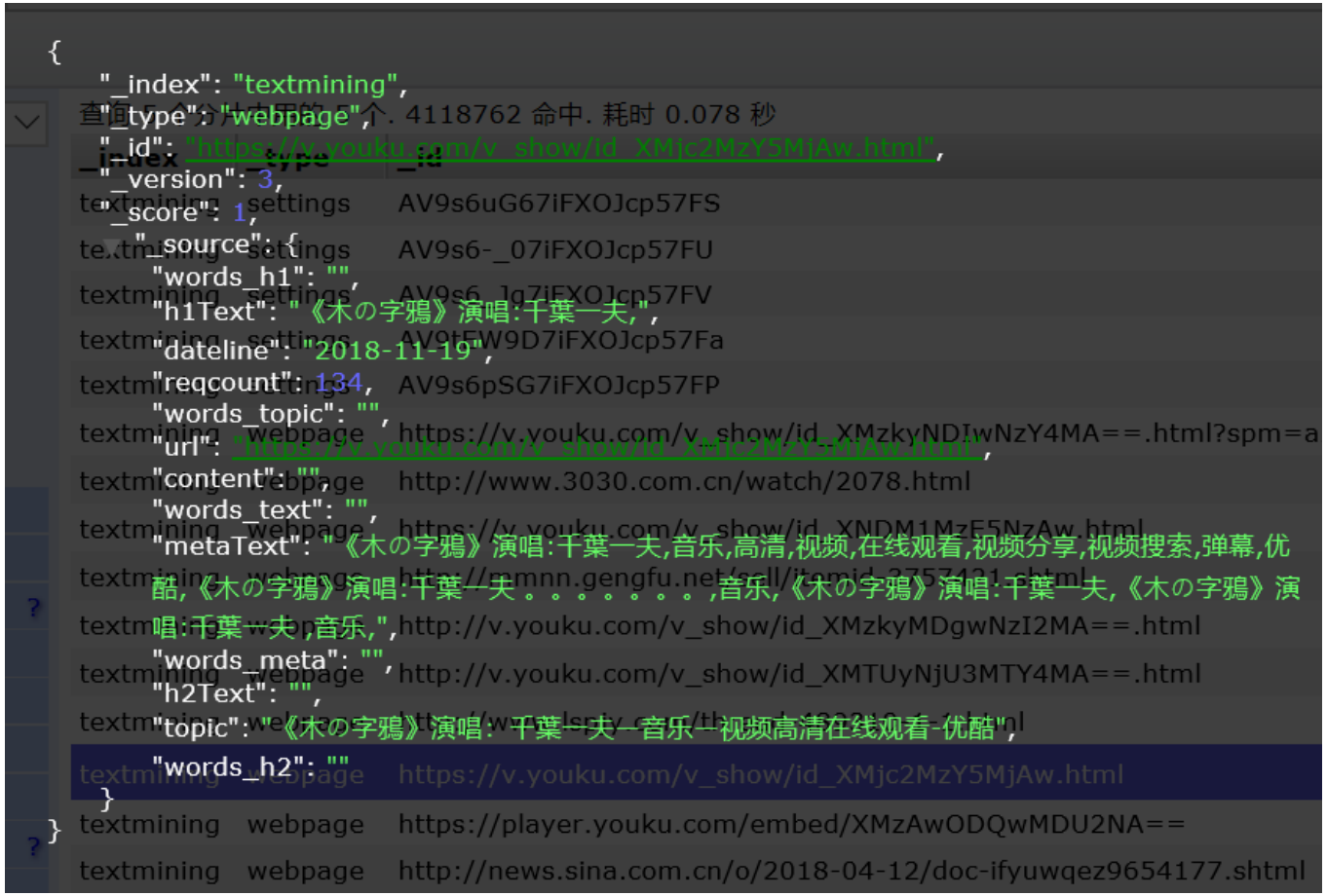
数据类型

- Index：索引，由很多的Document组成
- Document：由很多的Field组成，是Index和Search的最小单位。
- Field：由很多的Term组成，包括Field Name和Field Value
- Term：由很多的字节组成。一般将Text类型的Field Value分词之后的每个最小单元或者Keyword类型的Field Value叫做Term，一般来说，一个Term占据倒排索引的一行数据

数据类型

docid	name	age	id
1	Alice	18	101
2	Alice	20	102
3	Alice	21	103
4	Alan	21	104
5	Alan	18	105

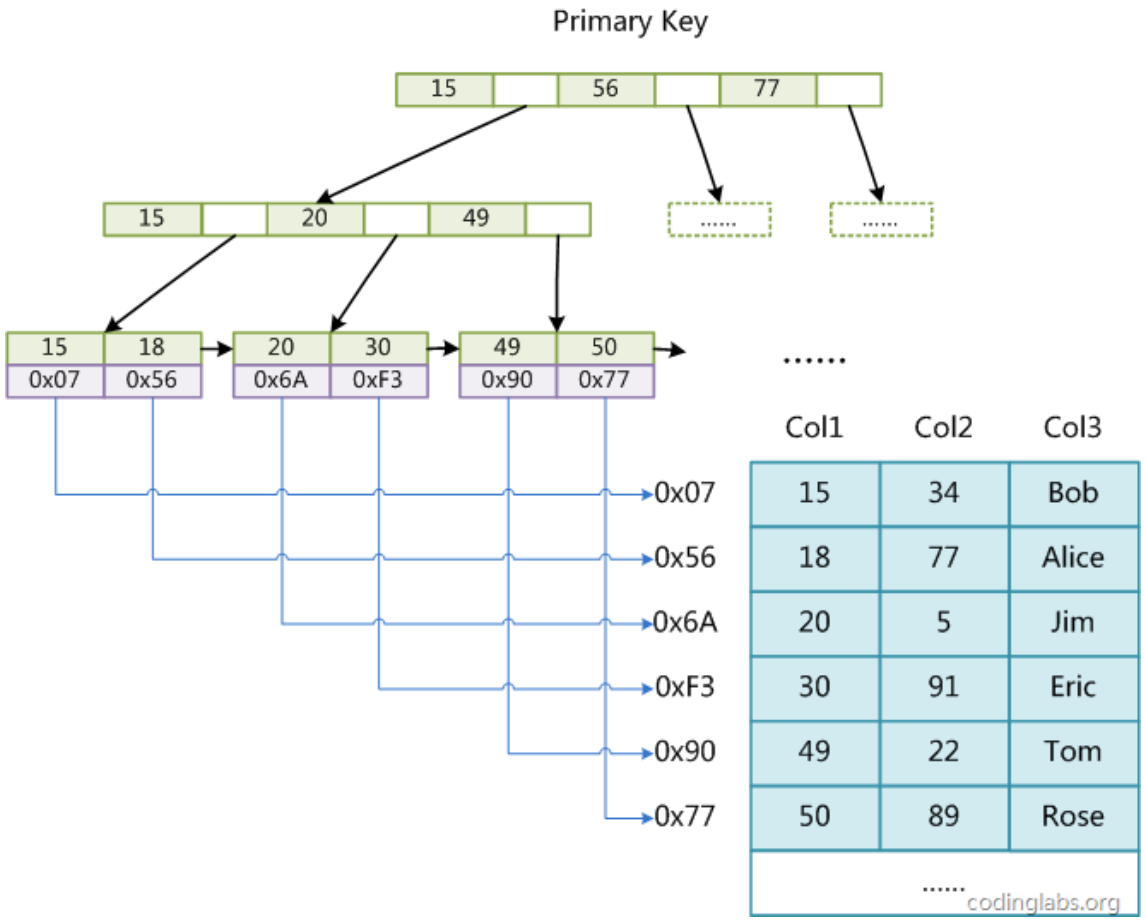
ES原始数据


{
"_index": "textmining",
"_type": "webpage",
"_id": "https://v.youku.com/v_show/id_XMjc2MzY5MjAw.html",
"_version": 3,
"_score": 1,
"_source": {
"words_h1": "",
"h1Text": "《木の字鴉》演唱:千葉一夫,",
"dateline": "2018-11-19",
"reqcount": 134,
"words_topic": "",
"url": "https://v.youku.com/v_show/id_XMjc2MzY5MjAw.html",
"content": "",
"words_text": "",
"metaText": "《木の字鴉》演唱:千葉一夫,音乐,高清,视频,在线观看,视频分享,视频搜索,弹幕,优酷,《木の字鴉》演唱:千葉一夫。。。。。。。。,音乐,《木の字鴉》演唱:千葉一夫,《木の字鴉》演唱:千葉一夫,音乐,",
"words_meta": "",
"h2Text": "",
"topic": "《木の字鴉》演唱:千葉一夫一音乐一视频高清在线观看-优酷",
"words_h2": ""
}
}

ES Mapping结构


```
{"textmining":

{"mappings":{"webpage":
    {"properties":
        {"content":{"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},

        "dateline":{"type":"date"},

        "h1Text":{"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},

        "h2Text":{"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},

        "metaText":{"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},"query":{"properties":{"range":{"properties":{"dateline":{"properties":{"lte":{"type":"date"}}}}}}},

        "reqcount":{"type":"long"},

        "topic":{"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},

        "url":{"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},

        "words_h1":{"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},

        "words_h2":{"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},

        "words_meta":{"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},

        "words_text":{"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},

        "words_topic":{"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}
}}}}}}}
```

典型B+Tree索引



内节点不存储data，只存储key；叶节点的data域存放的是数据记录的地址

为了提高查询的效率，减少磁盘寻道次数，将多个值作为一个数组通过连续区间存放，一次寻道读取多个数据，同时也降低树的高度

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上

倒排结构

一种预处理技术，避免穷举搜索

Inverted Index 101

Query: keeper

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

term	freq	
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>

Dictionary and posting lists

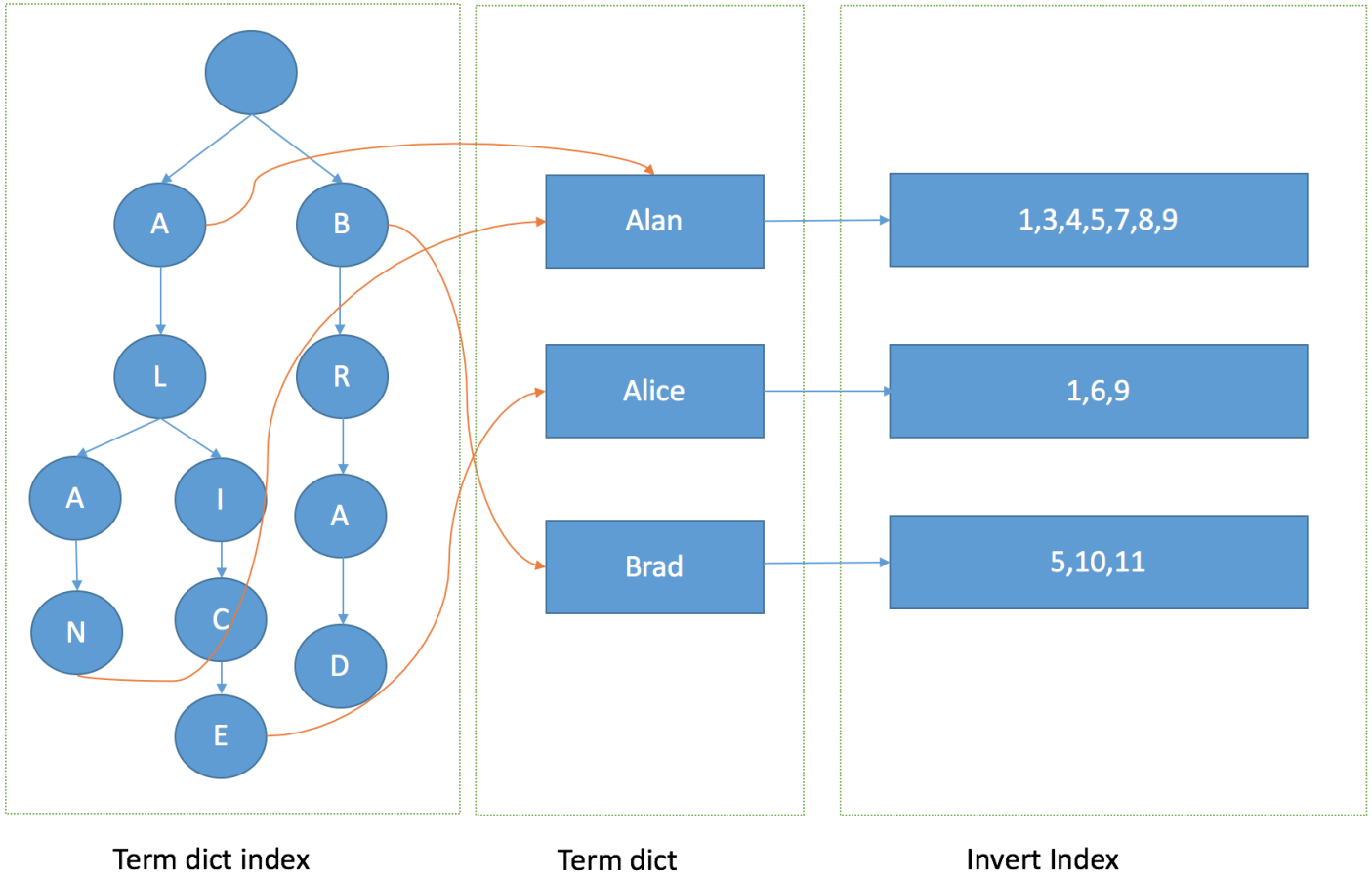
倒排结构

- 倒排本质上就是基于term的反向列表，方便进行文档id查找。如果term非常多，如何快速拿到这个倒排链呢？可以用二分查找的方式，比全遍历更快地找出目标的 term，这个就是 Term Dictionary
- B+Tree通过减少磁盘寻道次数来提高查询性能，Elastic Search 也是采用同样的思路，直接通过内存查找term，不读磁盘

倒排结构

- Term Dictionary里我们可以按照term进行排序，那么用一个二分查找就可以定为这个term所在的地址。这样的复杂度是O(logN)
- 如果term太多，Term Dictionary也会很大，放内存不现实，于是有了Term Index，就像字典里的索引页一样，A开头的有哪些term，分别在哪页，可以理解Term Index是一颗树

倒排结构

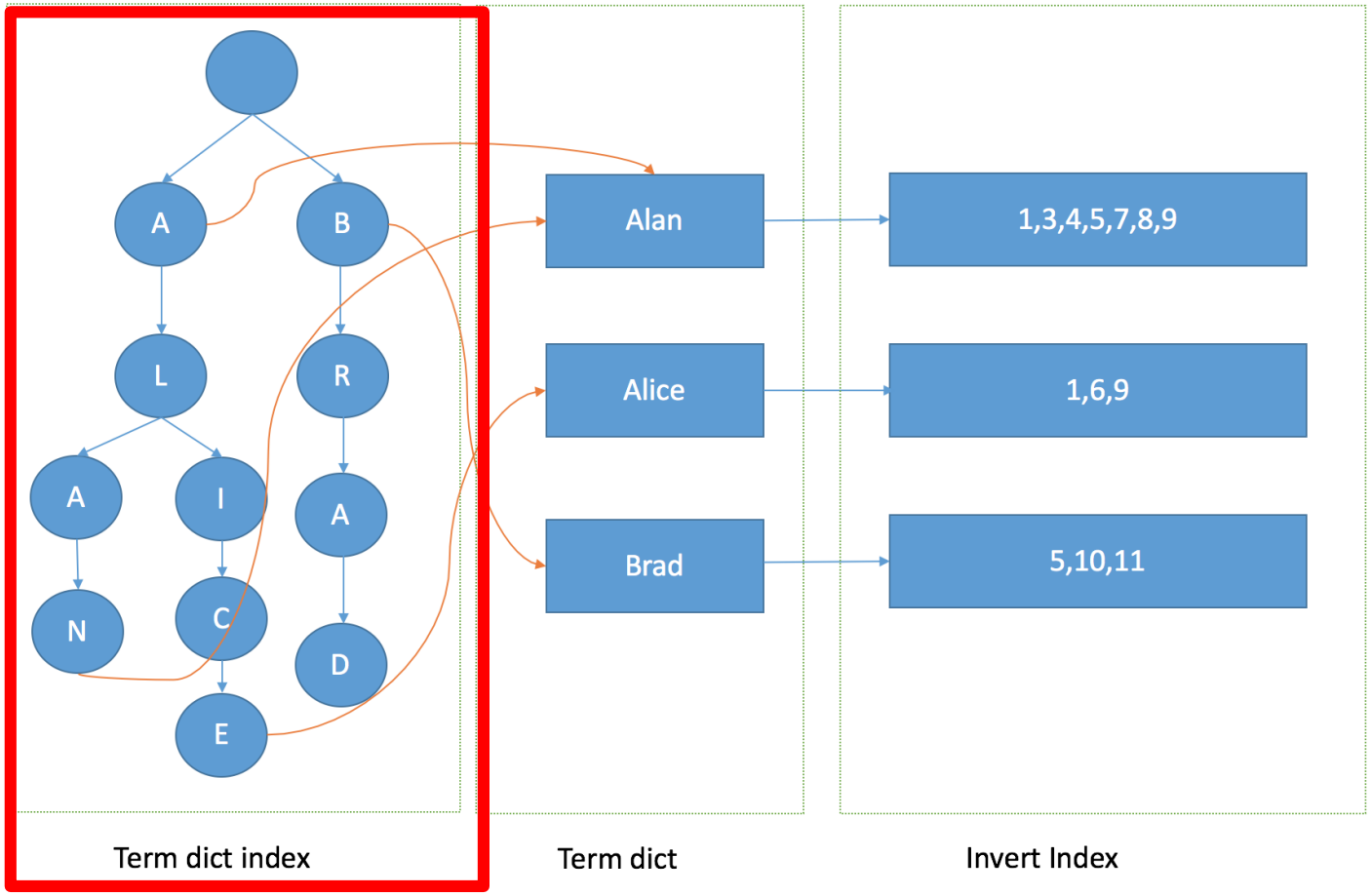


词索引不会包含所有的term，它包含的是term的一些前缀。

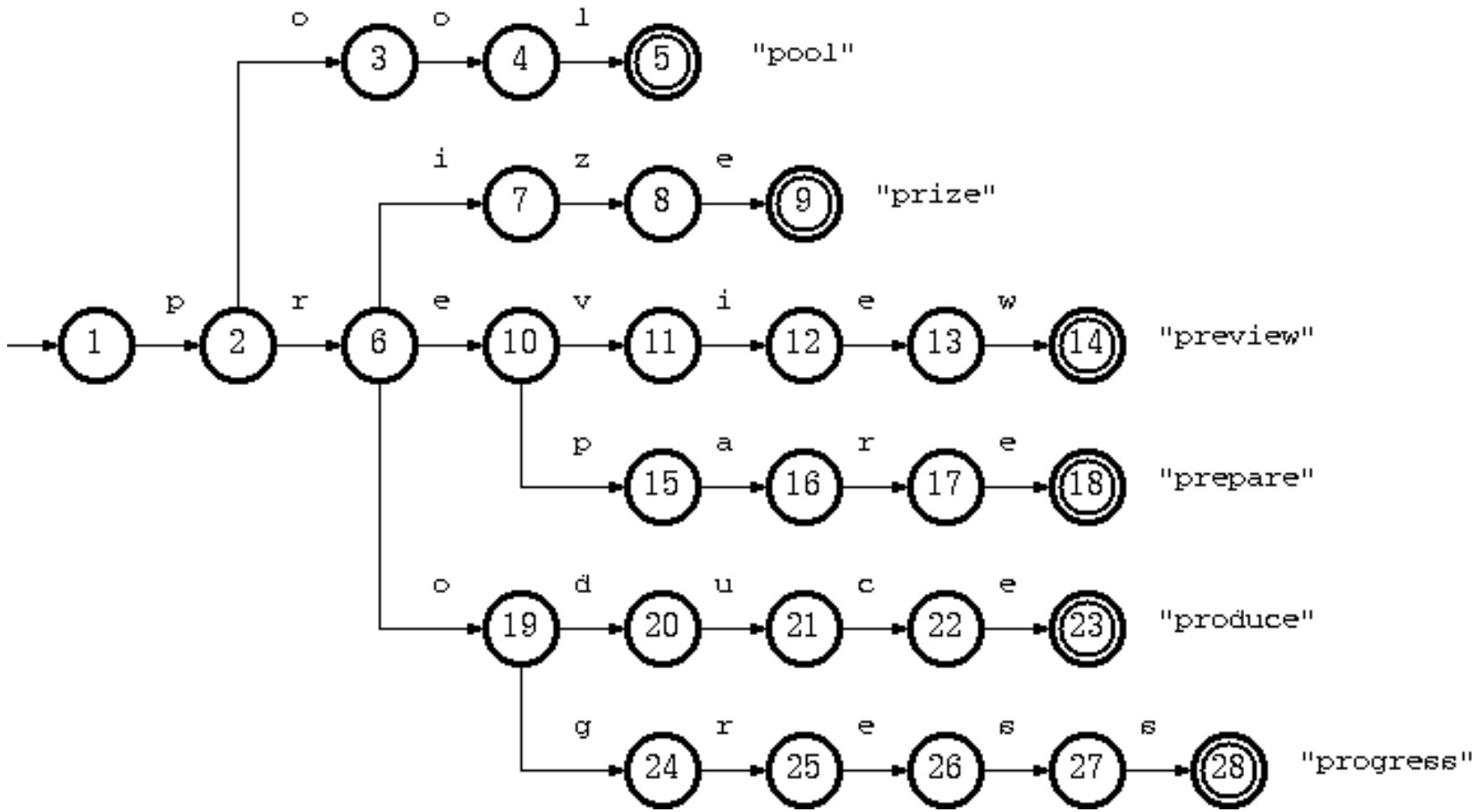
结合FST(Finite State Transducers)的压缩技术，可以使词索引缓存到内存中。

从词索引查到对应的词典的block位置之后，再去磁盘上找词，大大减少磁盘随机读的次数

倒排结构


A
B
L
R
A
I
A
N
C
D
E
Alan
Alice
Brad
1,3,4,5,7,8,9
1,6,9
5,10,11
Term dict index
Term dict
Invert Index

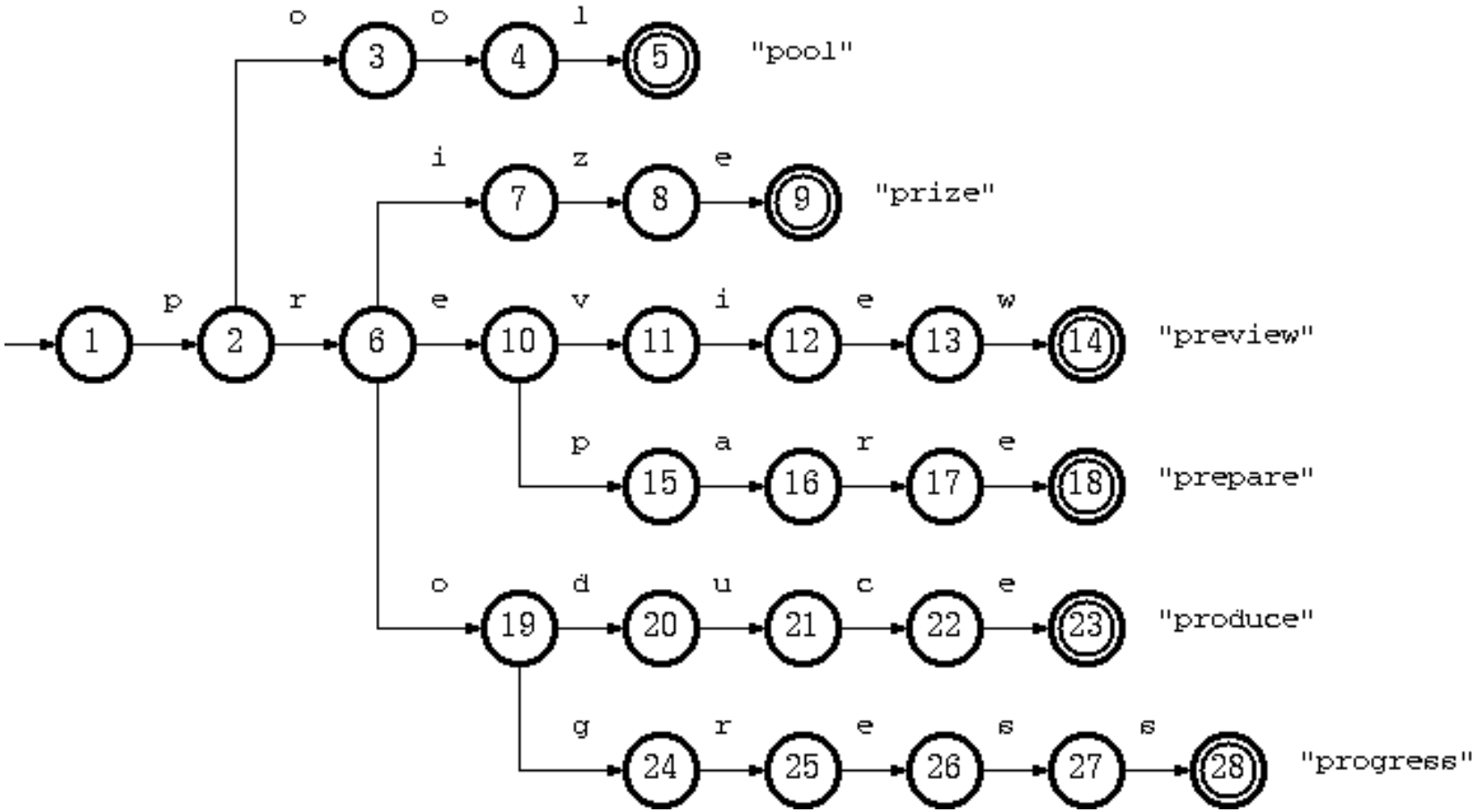
Trie



Trie树也称字典树，能在常数时间O(1)内实现插入和查询操作，是一种以空间换取时间的数据结构，广泛用于词频统计和输入统计领域

适合英文词典，如果系统中存在大量字符串且这些字符串基本没有公共前缀，则相应的Trie树将非常消耗内存

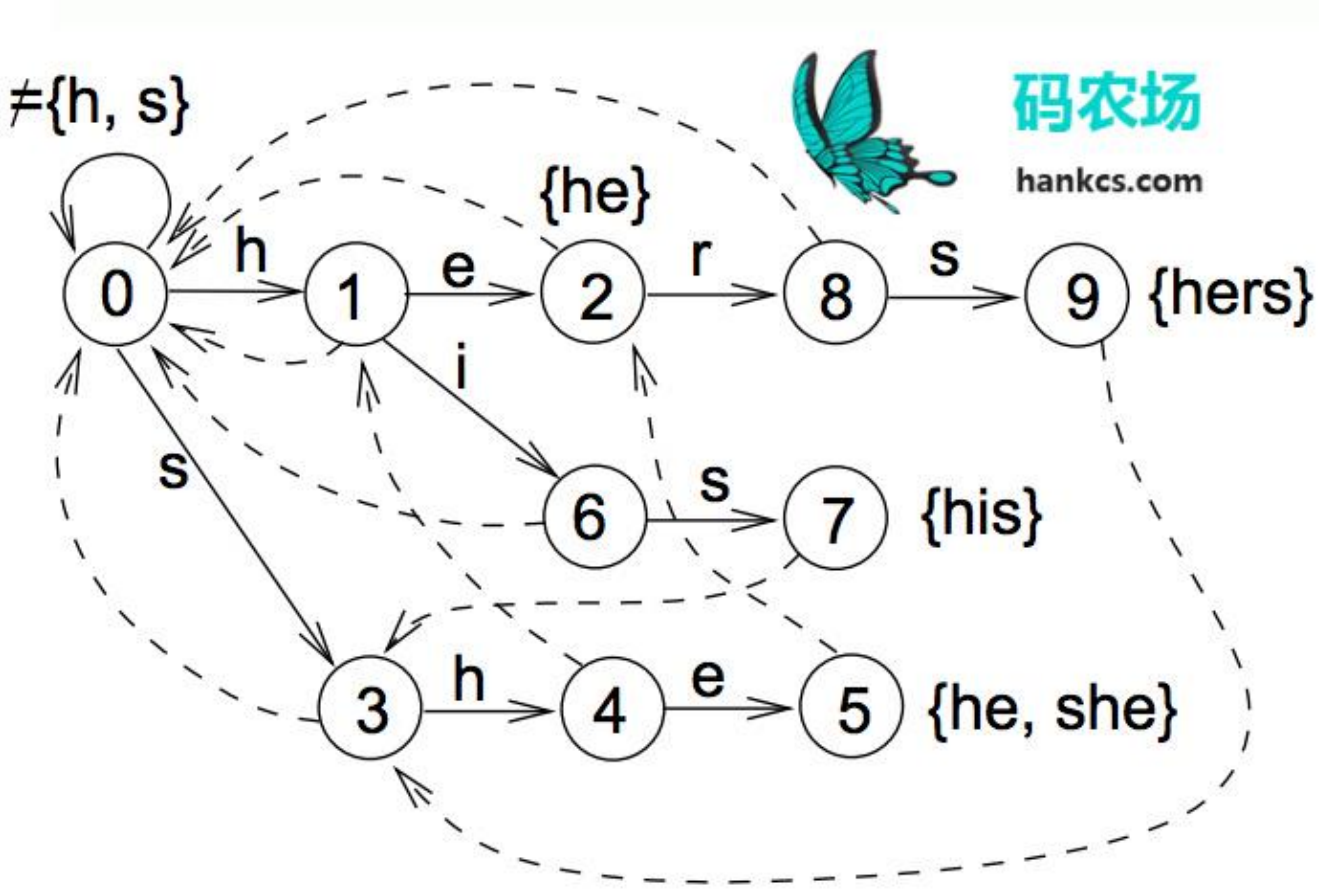
Double Array Trie



双数组Trie树能高速O(n)完成单串匹配，并且内存消耗可控，然而软肋在于多模式匹配

如果要匹配多个模式串，必须先实现前缀查询，然后频繁截取文本后缀才可多匹配，这样一份文本要回退扫描多遍，性能极低

Aho Corasick Automata



AC自动机能高速完成多模式匹配，然而具体实现聪明与否决定最终性能高低。

大部分实现都是一个Map<Character, State>了事，无论是TreeMap的对数复杂度，还是HashMap的巨额空间复杂度与哈希函数的性能消耗，都会降低整体性能

AC自动机

The Net Complexity

- Our preprocessing time is
  - $\Theta(n)$ work to build the trie,
  - $O(n)$ work to fill in suffix links, and
  - $O(n)$ work to fill in output links.
- Total preprocessing time: $\Theta(n)$.

The Final Totals

- We now have a multi-string search data structure with time complexity

$$\langle O(n), O(m + z) \rangle.$$

如果能用双数组Trie树表达AC自动机，就能集合两者的优点，得到一种近乎完美的数据结构

在O(n)时间内预处理和搜索，但是占据存储空间太大

常用来进行基于词典的分词

词典分词就是按照句子的字符顺序从根节点往下走，每走到一个结束节点则分出一个词

现在用的也不多了

Finite State Transducers

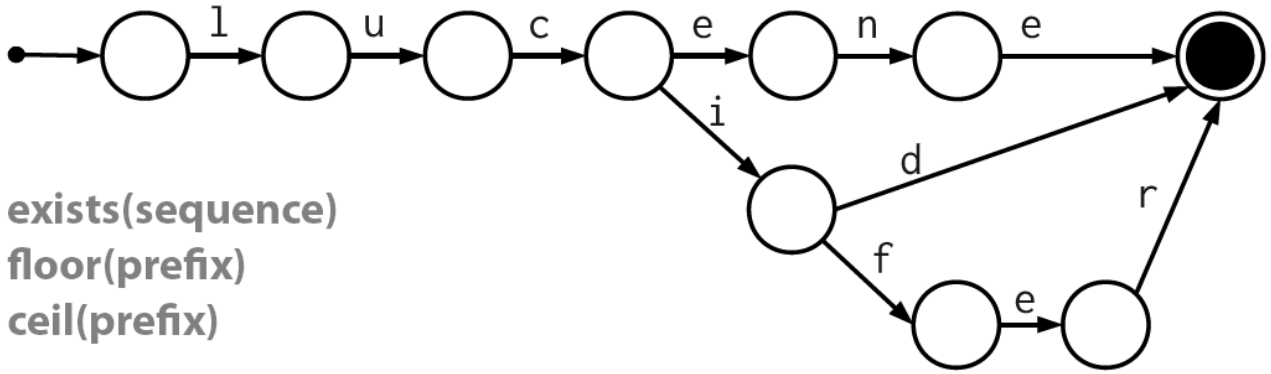
- 可以用一个Hashmap，当有一个term进入，hash继续查找倒排链，这里Hashmap的方式可以看做是term dictionary的一个index，性能高，但是内存消耗大，几乎是原始数据的三倍
- 从Lucene4开始，为了方便实现range query或者前缀，后缀等复杂的查询语句， Lucene使用FST数据结构来存储term字典

Finite State Automata

HashSet

hash	→ slot	→ value
0x29384d34		→ lucene
0xde3e3354		→ lucid
0x00000666		→ lucifer

FSA (deterministic)



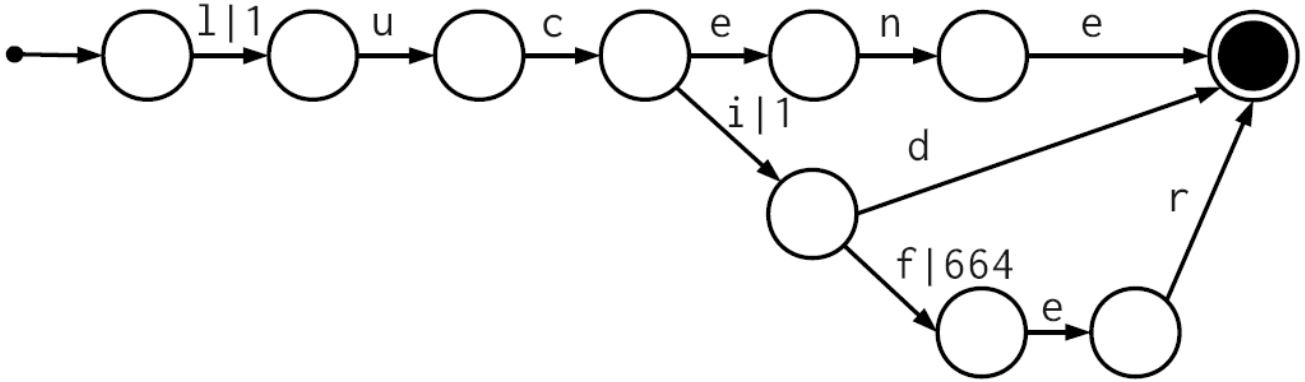
exists(sequence)
floor(prefix)
ceil(prefix)

Finite State Transducers

(Sorted)Map

```
lucene   → 1
lucid    → 2
lucifer  → 666
```

FST (transducer)



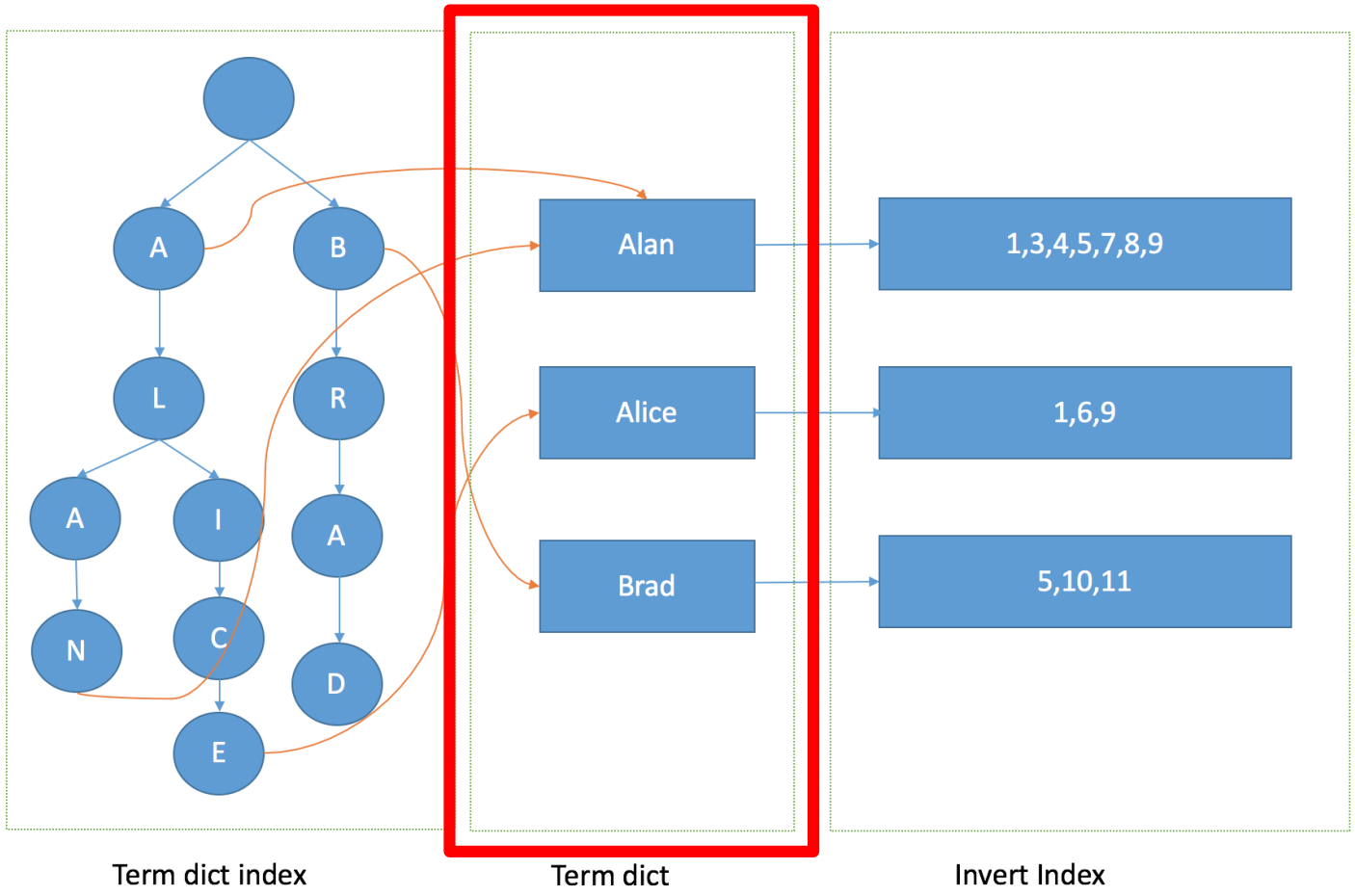
Finite State Transducers

- 空间占用小，通过对词典中单词前缀和后缀的重复利用，压缩了存储空间
- FST压缩率一般在3倍~20倍之间，相对于TreeMap/HashMap的膨胀3倍，内存节省就有9倍到60倍
- 查询速度快，O(len(str))的查询时间复杂度

Finite State Transducers

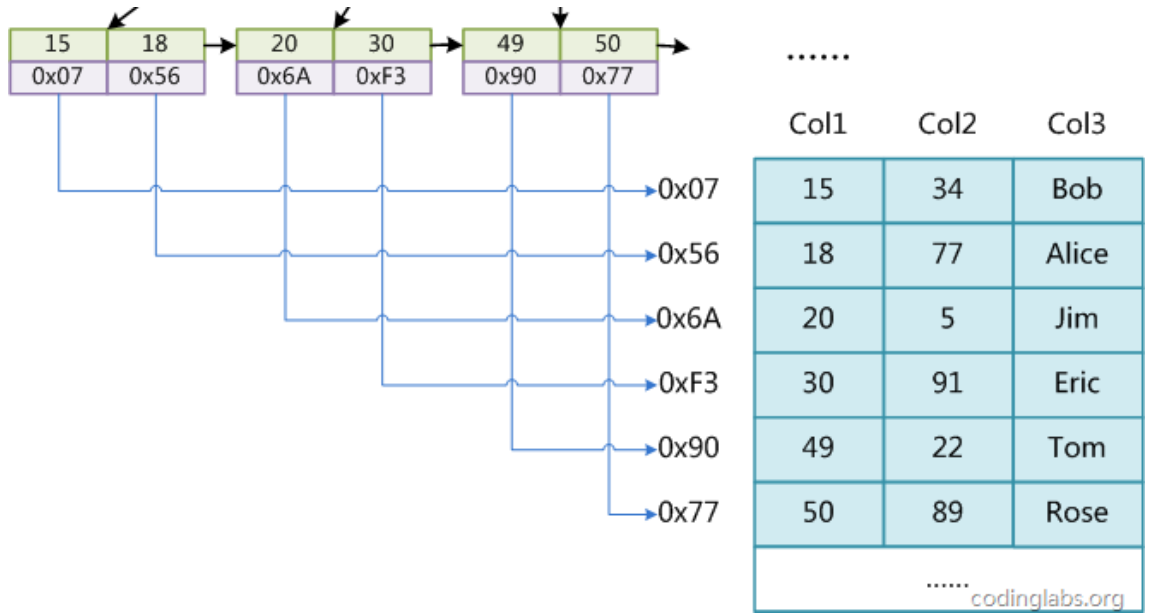
- FST在单term查询上可能相比Hashmap并没有明显优势，甚至会慢一些，但是在范围，前缀搜索以及压缩率上都有明显的优势
- 在通过FST定位到倒排链后，有一件事情需要做，就是倒排链的合并。因为查询条件可能不止一个，例如上面我们想找name="alan" and age="18"的列表
- Lucene是如何实现倒排链的合并呢，需要看一下倒排链存储的数据结构

倒排结构


A
B
L
R
A
I
A
N
C
D
E
Alan
Alice
Brad
1,3,4,5,7,8,9
1,6,9
5,10,11
Term dict index
Term dict
Invert Index

BlockTree terms dictionary

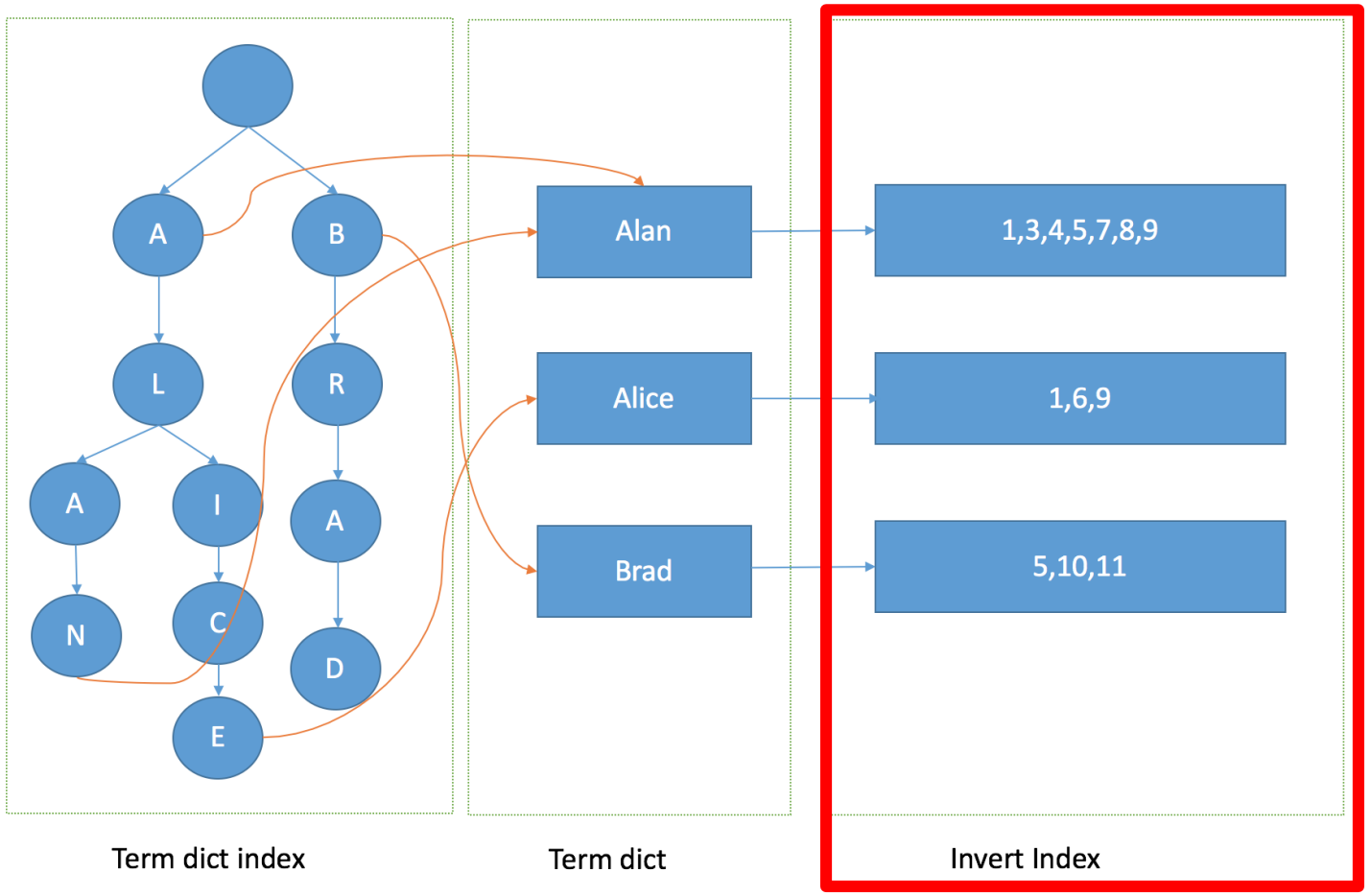
- The purpose of the terms dictionary is to store all unique terms seen during indexing, and map each term to its metadata (docFreq, totalTermFreq, etc.), as well as the postings (documents, offsets, postings and payloads). When a term is requested, the terms dictionary must locate it in the on-disk index and return its metadata.



.....

Col1	Col2	Col3
15	34	Bob
18	77	Alice
20	5	Jim
30	91	Eric
49	22	Tom
50	89	Rose
.....		

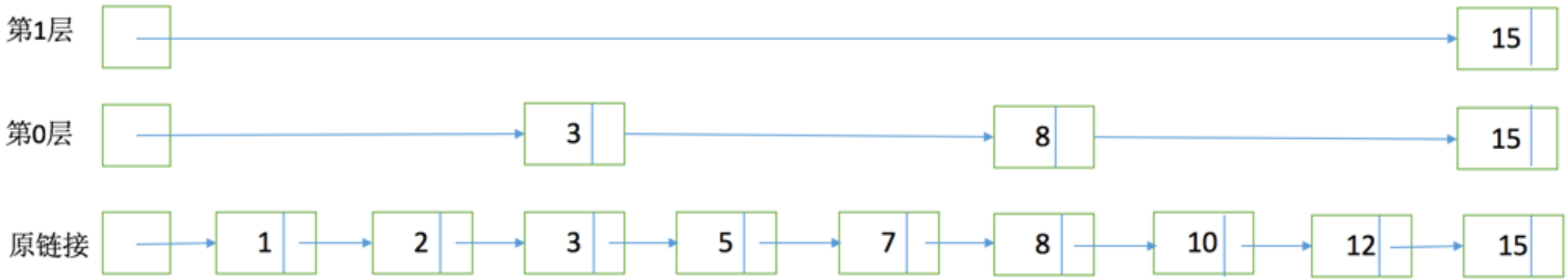
倒排结构


A
B
L
R
A
I
A
N
C
D
E
Alan
Alice
Brad
1,3,4,5,7,8,9
1,6,9
5,10,11
Term dict index
Term dict
Invert Index

SkipList

为了能够快速查找docid，lucene采用了SkipList这一数据结构。SkipList有以下几个特征：

1. 元素排序的，对应到我们的倒排链，lucene是按照docid进行排序，从小到大。
2. 跳跃有一个固定的间隔，这个是需要建立SkipList的时候指定好，例如下图以间隔是3
3. SkipList的层次，这个是指整个SkipList有几层



Linked List

Linked Lists Benefits & Drawbacks

- Benefits:
  - Easy to insert & delete in O(1) time
  - Don't need to estimate total memory needed
- Drawbacks:
  - Hard to search in less than O(n) time (binary search doesn't work, eg.)
  - Hard to jump to the middle

常数时间内插入和删除

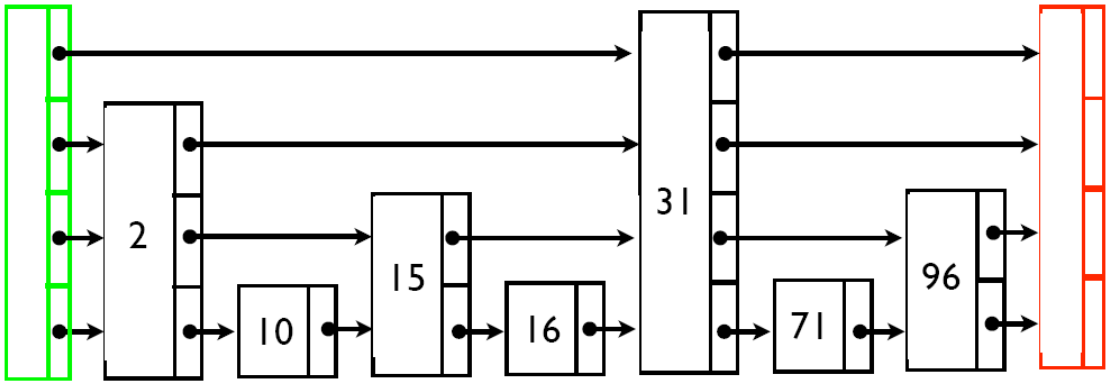
O(n)时间搜索

Perfict Skip List

Perfect Skip List

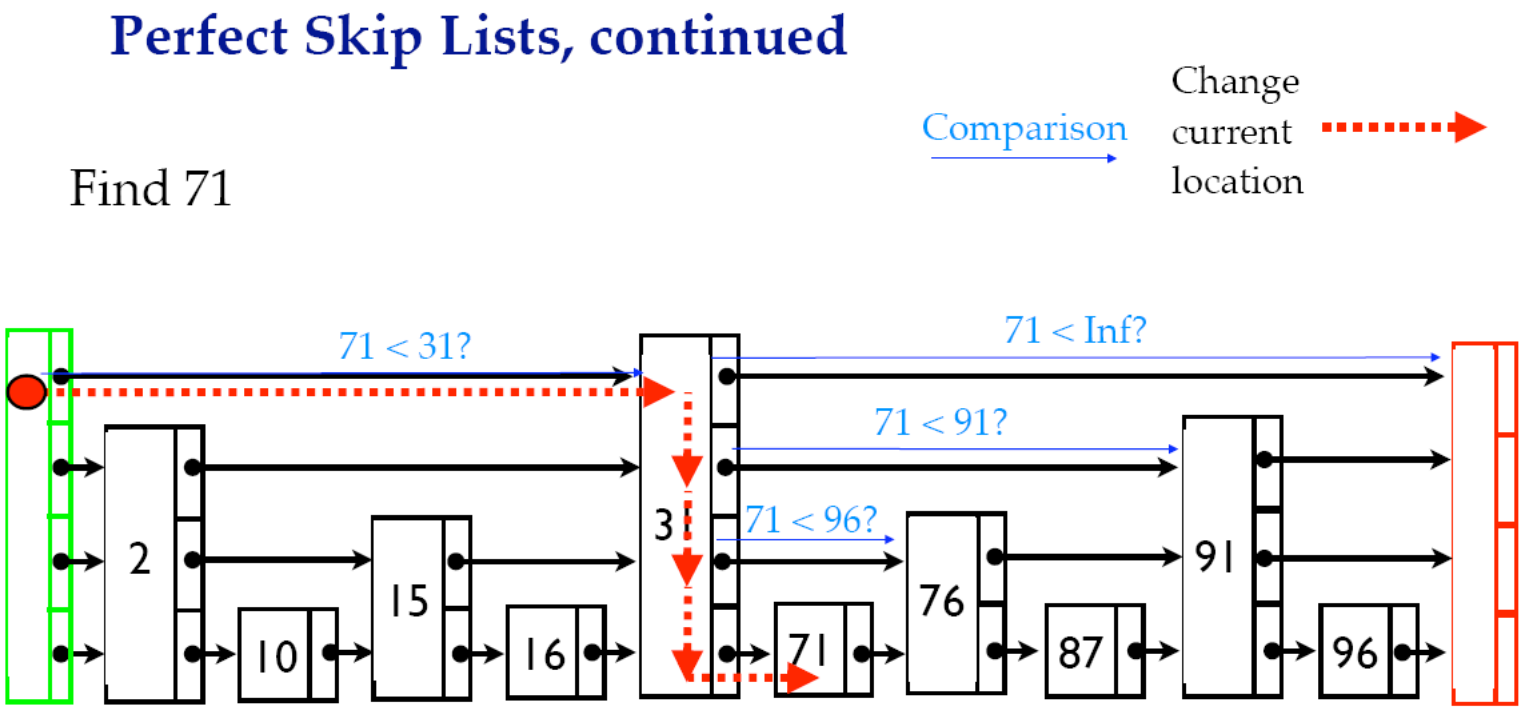
Perfect Skip Lists

- Keys in sorted order.
- O(log *n*) *levels*
- Each higher level contains 1/2 the elements of the level below it.
- Header & sentinel nodes are in every level



Perfect Skip List在插入和删除数据时需要重新排列

Perfect Skip List



Perfict Skip List

Search Time:

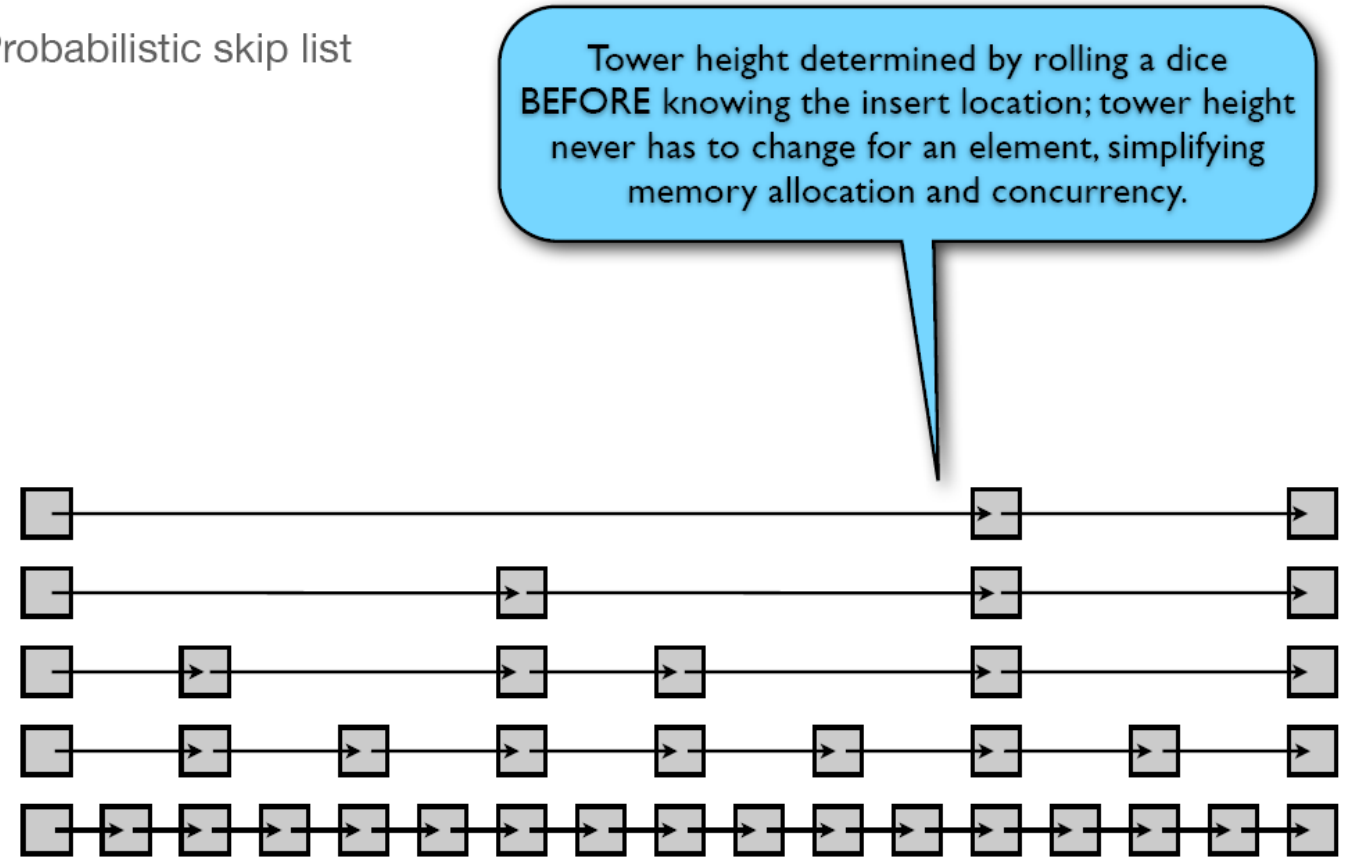
- O(log n) levels --- because you cut the # of items in half at each level

- Will visit at most 2 nodes per level:
  If you visit more, then you could have done it on one level higher up.

- Therefore, search time is O(log n).

Probabilistic Skip List

- Probabilistic skip list



实际中在插入新数据时，不会用完美跳表（插入中间位置需要重新调整链表），而是随机插入一个位置

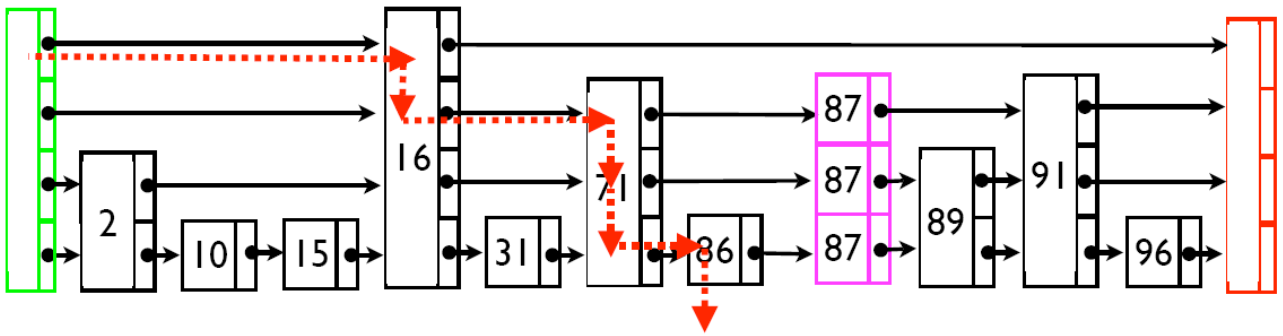
跳表相对于树结构而言比较容易实现

节点数目的期望值是 2n

Probabilistic Skip List

Insertion:

Insert 87



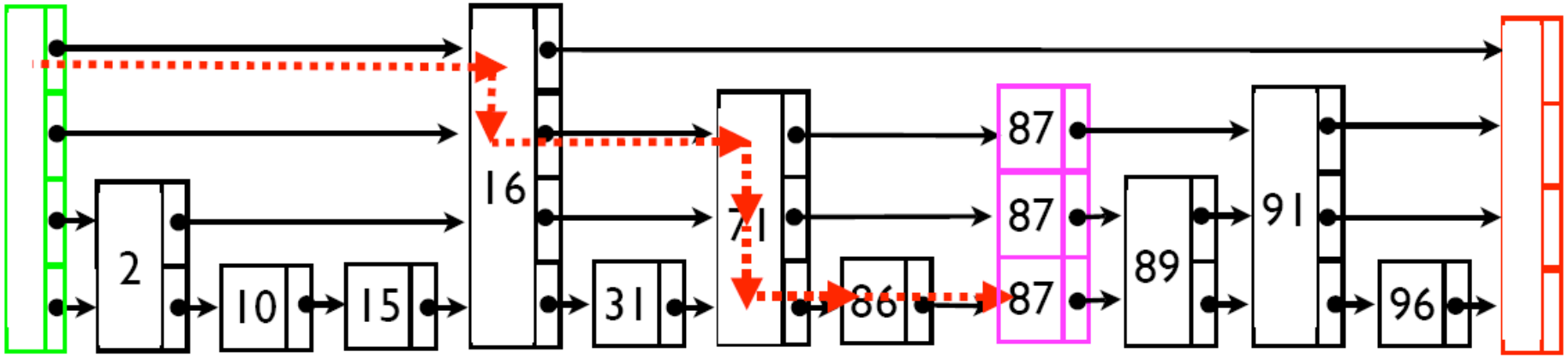

```
Find k
Insert node in level 0
let i = 1
while FLIP() == "heads":
    insert node into level i
    i++
```

Just insertion into a linked list after last visited node in level *i*

Probabilistic Skip List

Deletion:

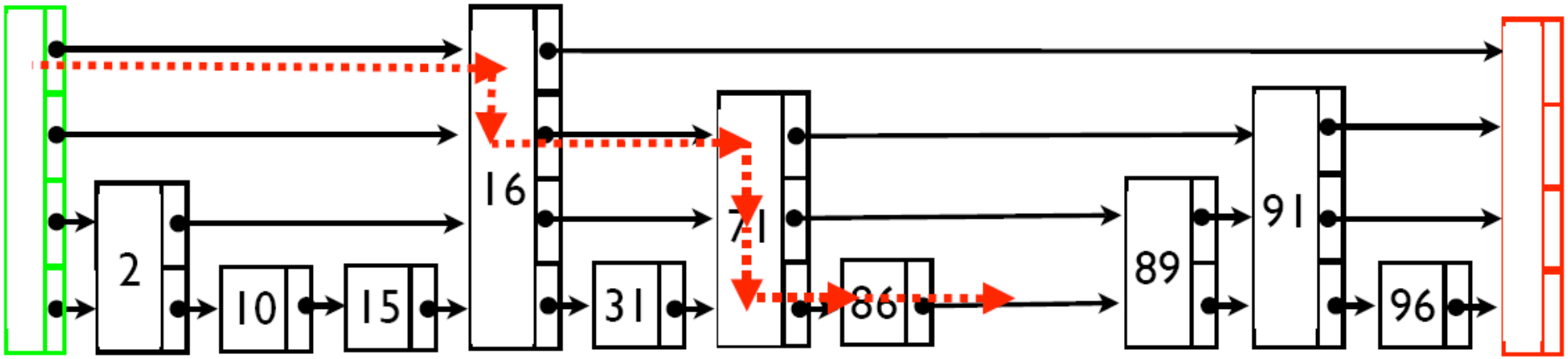
Delete 87



Probabilistic Skip List

Deletion:

Delete 87



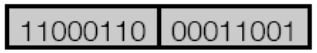
链表压缩

Posting list encoding

Doc IDs to encode: 5, 15, 9000, 9002, 100000, 100090

Delta encoding:

VInt compression:



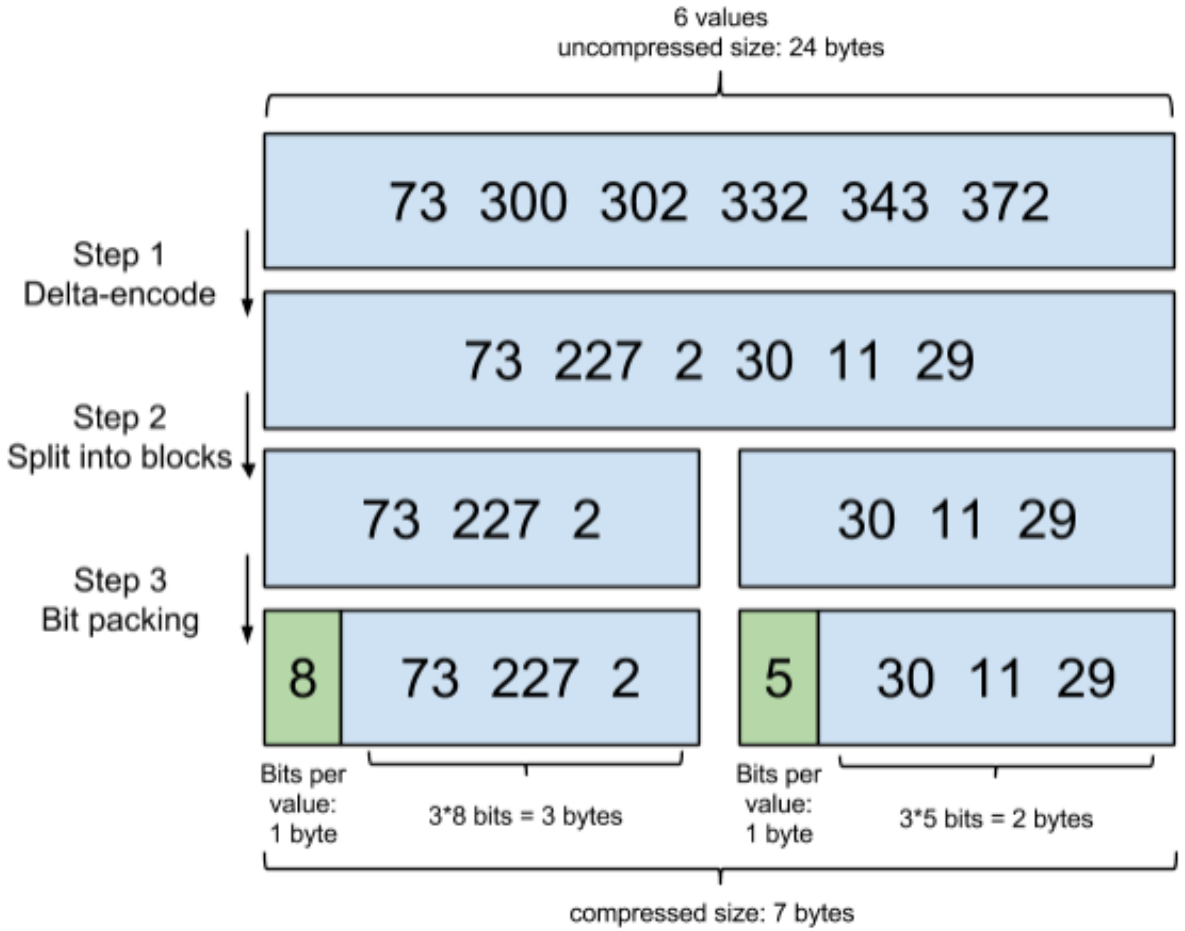
- Variable number of bytes - a VInt-encoded posting can not be written as a primitive Java type; therefore it can not be written atomically

两种压缩方式：

基于相对值的压缩

基于Bitset压缩

链表压缩-Frame Of Reference 编码

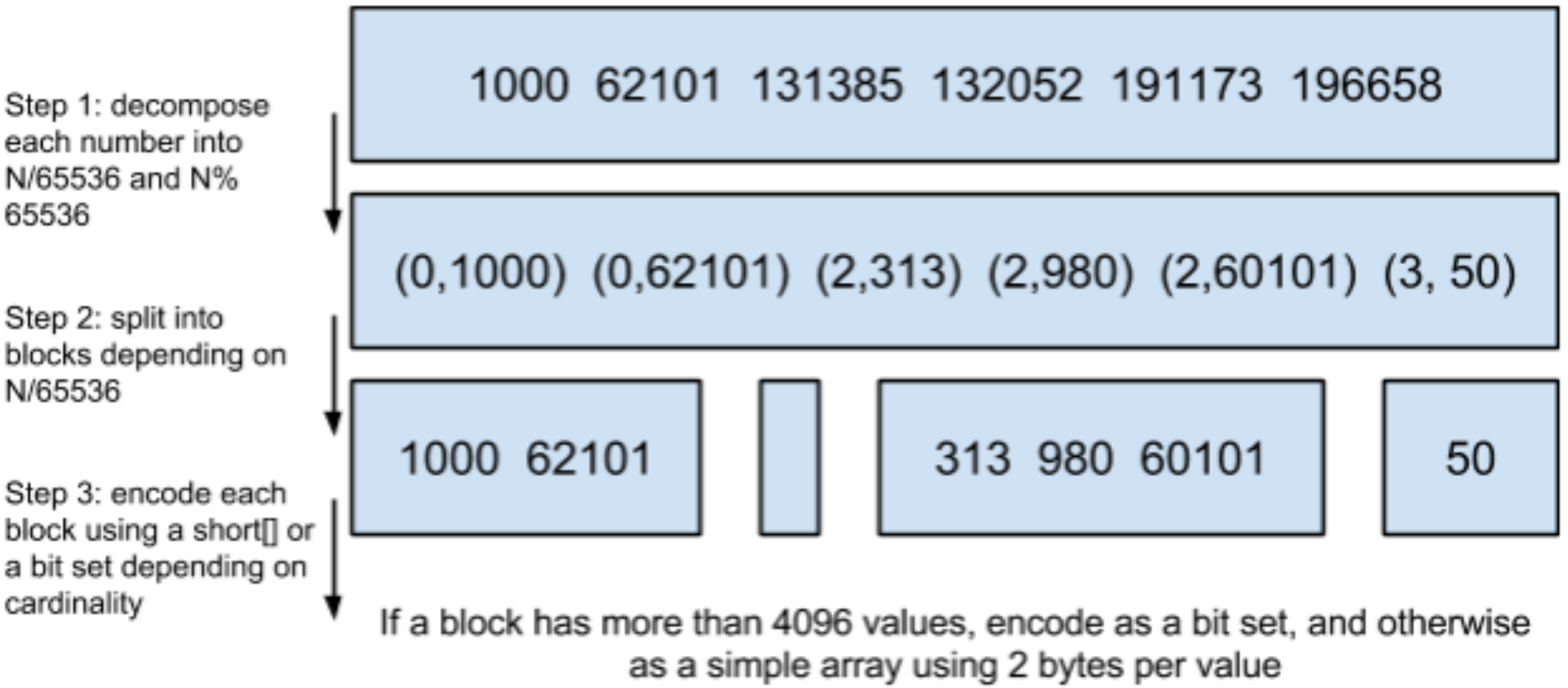


考虑到频繁出现的 term（所谓 low cardinality 的值），比如 gender 里的男或者女。如果有 1 百万个文档，那么性别为男的 posting list 里就会有 50 万个 int 值。

用 Frame of Reference 编码进行压缩可以极大减少磁盘占用。这个优化对于减少索引尺寸有非常重要的意义

存在的问题是只能从前往后读，也有相应的解决方法

链表压缩-利用 bitset 合并



每个文档按照文档 id 排序对应其中的一个 bit

Bitset 自身就有压缩的特点，其用一个 byte 就可以代表 8 个文档。所以 100 万个文档只需要 12.5 万个 byte

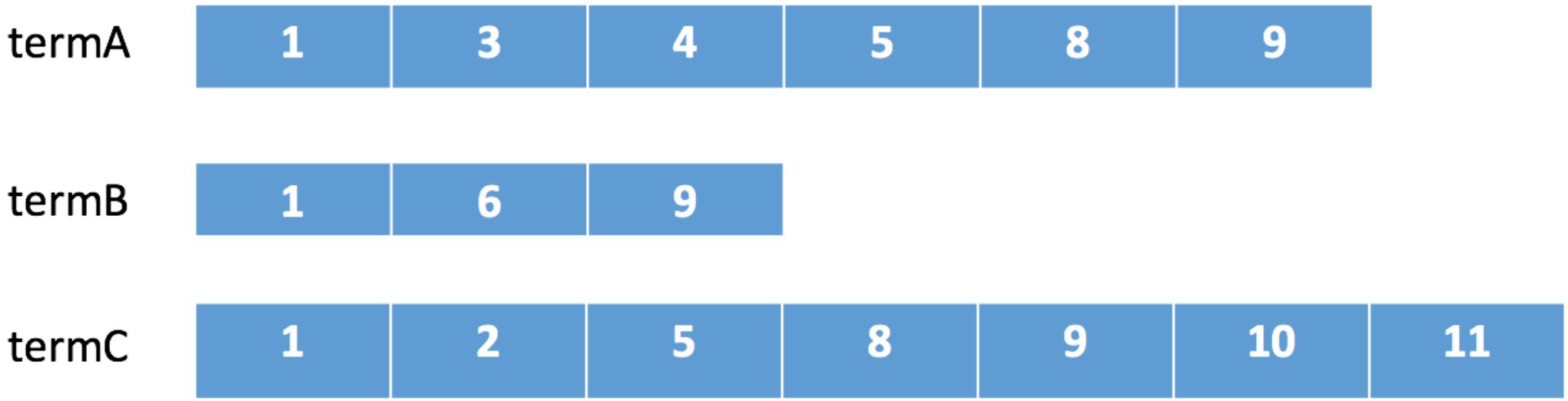
倒排合并

- 假如我们的查询条件是单个条件 name = “Alice”，那么按照之前的介绍，首先在term字典中定位是否存在这个term，如果存在的话进入这个term的倒排链，并根据参数设定返回分页返回结果即可。

倒排合并

- 假如我们有多个条件，例如我们需要按名字或者年龄单独查询，也需要进行组合 name = "Alice" and age = "18"的查询，那么使用传统二级索引方案，你可能需要建立两张索引表，然后分别查询结果后进行合并，这样如果age = 18的结果过多的话，查询合并会很耗时。
- 那么在lucene这两个倒排链是怎么合并呢。

倒排合并



倒排合并

- 如果查询的 filter 缓存到了内存中（以 bitset 的形式），那么合并就是两个 bitset 的 AND。如果查询的 filter 没有缓存，那么就用 skip list 的方式去遍历两个 on disk 的 skip list

Skip List-小结

- 如果某个链很短，会大幅减少比对次数，并且由于SkipList结构的存在，在某个倒排中定位某个docid的速度会比较快不需要一个个遍历，可以很快的返回最终的结果
- 从倒排的定位，查询，合并整个流程组成了lucene的查询过程，和传统数据库的索引相比，lucene合并过程中的优化减少了读取数据的IO，倒排合并的灵活性也解决了传统索引较难支持多条件查询的问题

索引文档-小结

- 倒排结构
- FST
- Skip List
- 链表压缩
- BKD-Tree

Elastic Search-查询文档

Recall和Precision

Precision is the number of relevant documents retrieved divided by the total number of retrieved documents.

Precision = relevant documents retrieved / retrieved documents

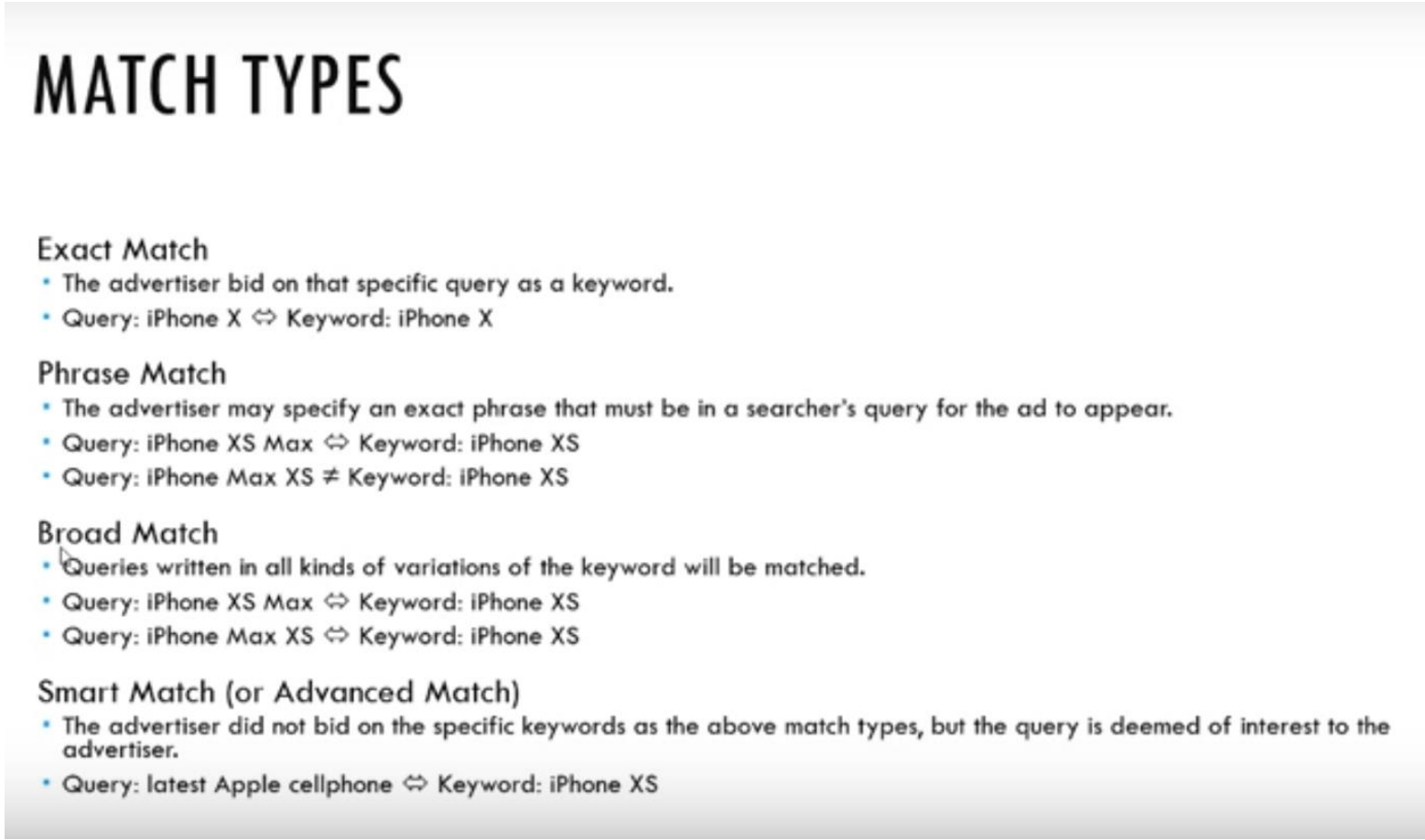
Recall is the number of relevant documents retrieved divided by the total of all relevant documents.

Recall = relevant documents retrieved / relevant documents

Recall和Precision

- 准确度=检索到的相关文档数量/检索到的文档总数
- 召回率=检索到的相关文档数量/所有相关文档

搜索广告的匹配类型



精确匹配：文档等于查询词

短语匹配：文档包含查询词

广泛匹配：查询词和文档的距离在一定范围之内

提升Recall

- Match all
- Simple is more
- Expand matches via analyzed fields that contain stemmers and synonyms
- Query across multiple fields
- Go for maximum disjunction
- Get fuzzy-wuzzy

Match查询

- match is used for full text searches in any field (use multi_match for multiple fields)
- 对h1text和query进行分词，并会根据score进行排序，速度会慢很多
- {
  "query":
  {"match": {"h1Text": "防范借贷风险"}
  }
  }
- Documents retrieved will be scored using the Practical Scoring Function

Multi Match查询

- 允许匹配一个或多个查询词
- 如果在一个文档中找到查询词的多个匹配，那个这个文档将会被赋予更高的分值

Multi Match查询

```
{
  "query":
    {"multi_match":  {
        "fields": [
            "h1Text",
            "h2Text"
        ],
        "query": "借贷",
        "operator": "or"
     }
  }
}
```

Match_Phrase查询

- 以下是contains匹配，类似于SQL里面的like查询，query不会被分词
- {
  "query":
  {"match_phrase": {"h1Text": "借贷"}
  }
}

Fuzzy查询

- Fuzziness allows us to indicate how many edits can be made to a term that we would still consider to be a match
- Fuzziness is a technique to increase recall, but typically comes at a significant cost to precision.
- Hamming Distance(0101,1010)
- 打签过程中用得是Fuzzy查询
- URL覆盖量从Fuzzy查询改为Filter+Bool+Match_Phrase查询

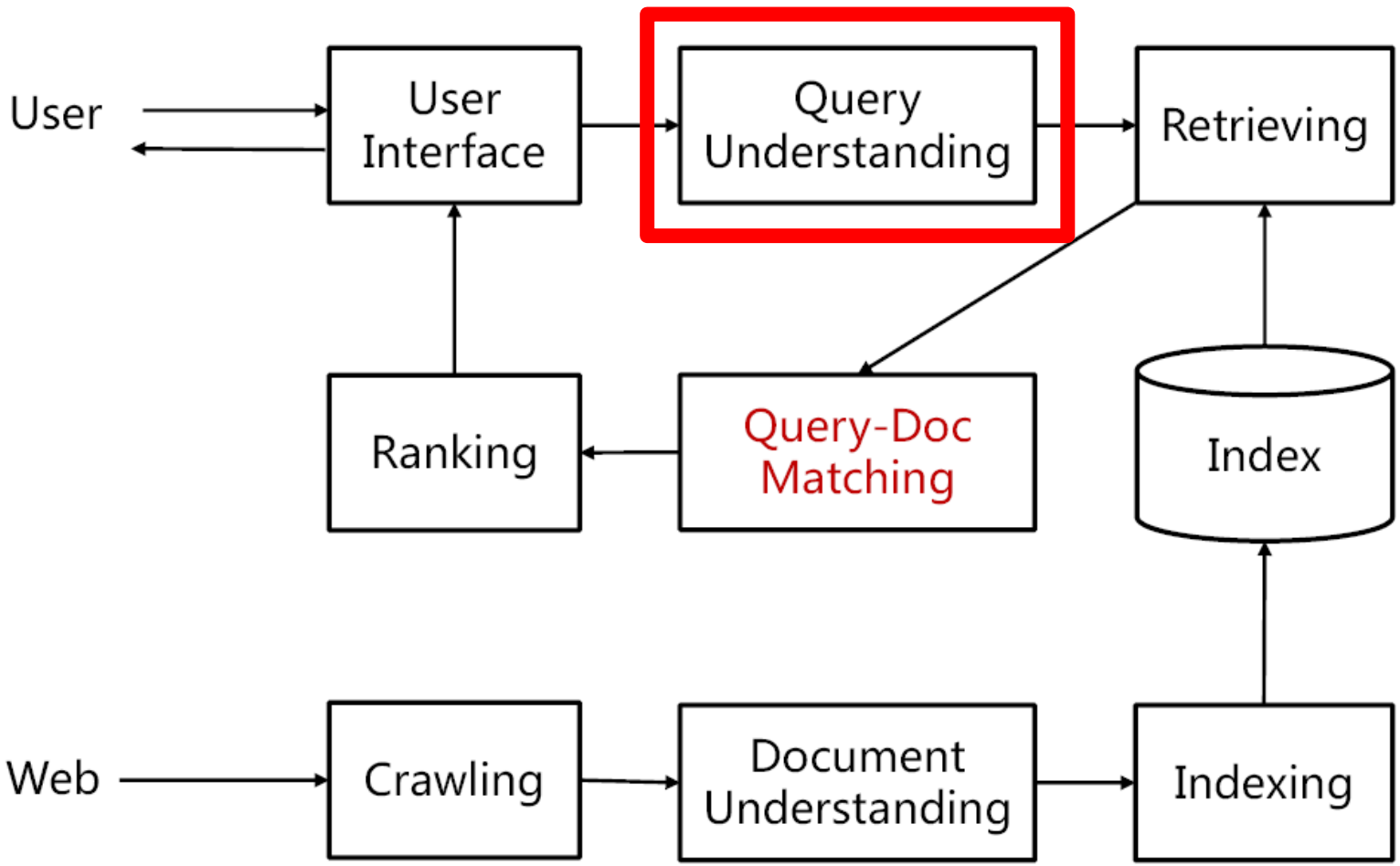
Fuzzy查询

```
{
  "query": {
    "fuzzy" : {
      "qualifications" : {
          "value" : "strategy",
          "fuzziness" : 2
      }
    }
  }
}
```

Fuzzy查询


```
{
  "query_string" : {
    "query" : "动漫 OR 游戏 OR 电影",
    "fields" : [
      "content^1.0",
      "h1Text^1.33",
      "h2Text^1.33",
      "metaText^1.67",
      "topic^1.67"
    ],
    "use_dis_max" : true,
    "tie_breaker" : 0.0,
    "default_operator" : "or",
    "auto_generate_phrase_queries" : true,
    "max_determinized_states" : 10000,
    "enable_position_increments" : true,
    "fuzziness" : "AUTO",
    "fuzzy_prefix_length" : 0,
    "fuzzy_max_expansions" : 50,
    "phrase_slop" : 0,
    "escape" : false,
    "split_on_whitespace" : true,
    "boost" : 1.0
  }
}
```

Overview of Web Search Engine



Query Understanding

1.1 QU/query understanding

1.1.1 概述

1.1.1.1 目的

- 拆解用户搜索词的意图
- 比如新品，年龄，尺码，属性，类目等搜索意图识别及归一

1.1.1.2 任务

- Query词性及主体结构，主要词/描述词等: 2018最新款适合胖胖的女生穿的**连衣裙**
- 预测用户搜索商品类目(category)性别(gender): 手提电脑、t恤 女
- 属性&标签识别: 品牌，颜色，尺寸: 裙子红色，43码nike球鞋
- 搜&逛:强意图/转化&弱意图/逛: 连衣裙 & Iphone XR 256G

1.1.1.3 方法

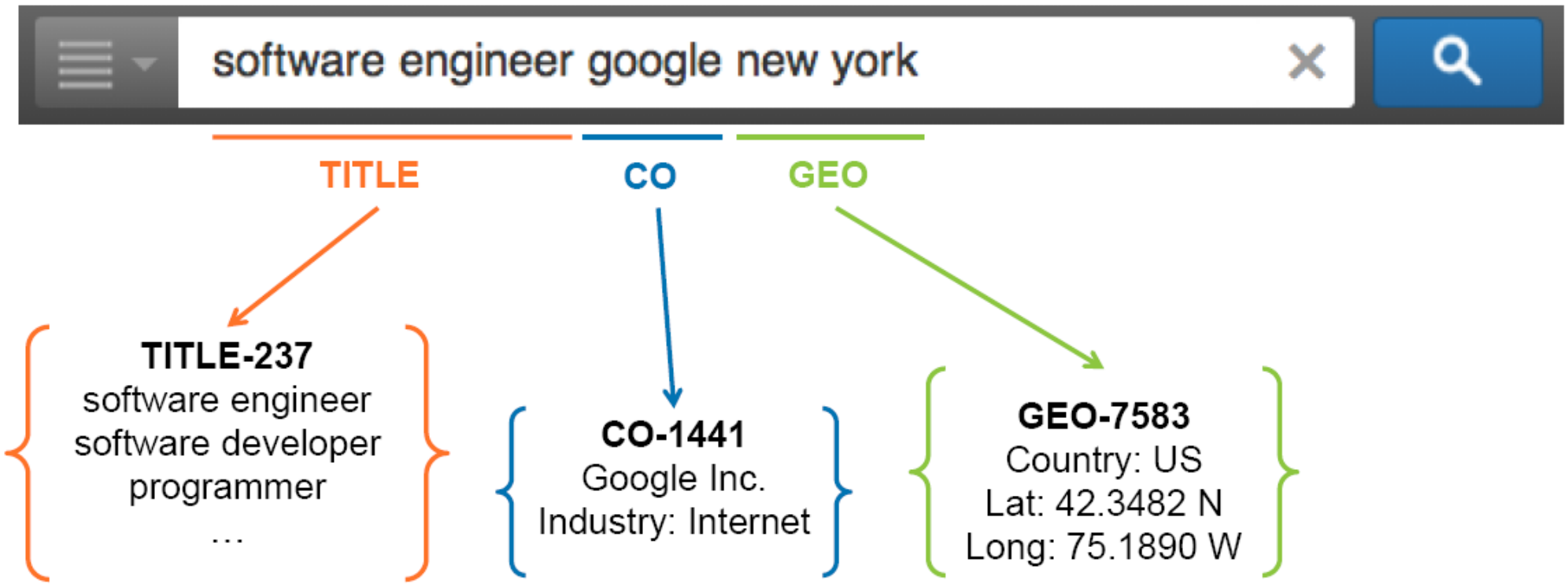
- 方法词表穷举法，规则解析法，机器学习方法

现实中关注更多的是查询理解、拓展、改写

利用拓展之后的词进行查询

Query Understanding可能包括Query Tagging实体标注和Query Expansion查询拓展

query tagging: identifying entities in the query



(RECOGNIZED TAGS: NAME, TITLE, COMPANY, SCHOOL, GEO, SKILL)

Next steps: query expansion

similar → MDR1000X is nice
similar → WH1000X is nice

word2vec
- words from vectors, given the context
- similarity-based synonyms
- classification
- little context ⇒ small accuracy

doc2vec: word2vec + doc dimension

Precision

- Define the rules
- Filter results
- Give your users an advanced search UI
- Set thresholds

单Term查询

- 以下是根据查询词精确匹配，term是代表完全匹配，即不进行分词器分析，Field中必须包含整个搜索的词汇
- {
  "query":
  {"term": {"h1Text.keyword": "防范借贷风险 南京中院发布百条提示,"}
  }
  }

Filter查询

- A filter performs an exact match in the specified field. Because of that, filters are typically used on not-analyzed fields (straight string, numeric, or date fields)
- 通常在非解析字段上进行查询
- There won't be any relevance ranking because all the documents are equal in terms of relevance
- 不会对搜索到的文档进行排序

Bool查询

- While performing the full text matching using analyzers where they exist, it also allows us to define rules for matching and the ability to combine queries (and filters) to better dial-in precision in the result set.
- 可以在bool查询基础上进行组合查询
- For boolean queries, we can define "must", "should", "should not", and "must not" matches.

minimum_should_match

- For example, we could set a minimum_should_match as 2, meaning that at least two of the conditional clauses in our boolean query have to match for us to consider the document a match for the query.

相关性评分

score(q,d) =

queryNorm(q)

* coord(q,d)

* SUM (

tf(t in d),

idf(t)²,

t.getBoost(),

norm(t,d)

) (t in q)

相关性评分和根据URL和关键词为人群打签过程中的签的得分有关

得分指的是关键词和文档的匹配程度

相关性评分

- queryNorm(q)是查询正则化因子
- coord(q,d)是文档中匹配到的检索词数量
- SUM循环内部计算的是以下项的和
  - tf(t in d)查询分词之后的在文档d中的检索词频率，描述的是这个词在多大程度上能够代表该文档
  - idf(t) 是检索词t的逆文档频率，描述的是词的总体重要程度
  - t.getBoost()是自定义的查询提升度
  - norm(t,d)是field的长度范数？？？

性能分析

- 对单个词条进行查询，Lucene会读取该词条的倒排链，倒排链中是一个有序的docId列表
- 对字符串范围/前缀/通配符查询，Lucene会从FST中获取到符合条件的所有Term，然后就可以根据这些Term再查找倒排链，找到符合条件的doc
- 对数字类型进行范围查找，Lucene会通过BKD-Tree找到符合条件的docId集合，但这个集合中的docId并非有序的

性能分析

- 总体上讲，扫描的doc数量越多，性能肯定越差。
- 单个倒排链扫描的性能在每秒千万级，这个性能非常高，如果对数字类型要进行Term查询，也推荐建成字符串类型。
- 通过Skip List进行倒排链合并时，性能取决于最短链的扫描次数和每次skip的开销，skip的开销比如BLOCK内的顺序扫描等。

性能分析

- FST相关的字符串查询要比倒排链查询慢很多(通配符查询更是性能杀手)
- 基于BKD-Tree的数字范围查询性能很好，但是由于BKD-Tree内的docID不是有序的，不能采用类似skipList的向后跳的方式，如果跟其他查询做交集，必须先构造BitSet，这一步可能非常耗时

小结

- 难点
  - Query Understanding
    - Entity Tagging 例如广告短语中的实体识别
      - CRF
    - Query Expansion
      - Word2Vec，Doc2Vec
      - 文本分类
  - Matching
    - 语义匹配

参考

- https://compose.com/articles/elasticsearch-query-time-strategies-and-techniques-for-relevance-part-i/
- https://compose.com/articles/elasticsearch-query-time-strategies-and-techniques-for-relevance-part-ii/
- https://www.compose.com/articles/how-scoring-works-in-elasticsearch/

参考

- Lucene 查询原理
- 基于Lucene查询原理分析Elasticsearch的性能
- 剖析Elasticsearch的IndexSorting:一种查询性能优化利器
- 时间序列数据库的秘密 (2)——索引

参考

- https://zhuanlan.zhihu.com/p/51201097
- https://www.youtube.com/watch?v=AQau4-VF64w&index=13&list=PLuRT8pAOHZrzGDI236hCtpM5NTgJNyvsm&t=0s
- http://yongyuan.name/blog/ann-search.html
- Product Quantizers for k-NN Tutorial part 1

参考

- http://yongyuan.name/blog/cbir-technique-summary.html
- MySQL索引背后的数据结构及算法原理
- https://github.com/facebookresearch/faiss
- http://mccormickml.com/2017/10/22/product-quantizer-tutorial-part-2/

参考

- 一次“高大上”模型的落地“失败”吐槽
- lucene字典实现原理
- http://www.hankcs.com/program/algorithm/aho-corasick-double-array-trie.html
- http://cs231n.github.io/transfer-learning/#tf

参考

- https://www.slideshare.net/CraigMacdonald/efficient-query-processing-infrastructures