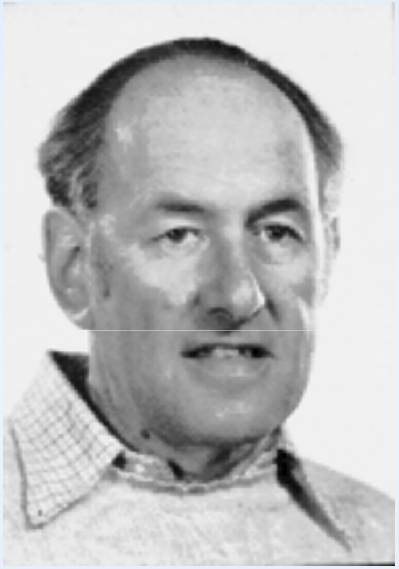
Vector Space Model of Salton
Salton传染媒介空间模型

Gerry Salton 1927-1995

Gerard A. "Gerry" Salton, was a Professor of Computer Science at Cornell University.

Salton was perhaps the leading computer scientist working in the field of information retrieval during his time, and "the father of Information Retrieval".

Wikipedia



Principle behind the Vector Space Model (VSM)
向量空间模型（VSM）背后的原理

One of the most widely used matching methods.

最广泛使用的匹配方法之一。

Each document is represented as a point on an *n*-dimensional sphere.

每个文档都表示为n维球体上的一个点。

Query is also a point on the same sphere.

查询也是同一球体上的一个点。

The document points close to the query point are retrieved, ranked so that the closest document is first and the furthest document last.

检索接近查询点的文档点，对其进行排序，使得最接近的文档是第一个，最远的文档是最后的。

Vector Interpretation of the VSM
矢量解释VSM

There is a term vector for each document.

每个文档都有一个术语向量。

Each element in the vector corresponds to a term.

向量中的每个元素对应一个术语。

The value for a term is its TF*IDF score.

术语的值是其TF * IDF分数。

The vector is first normalised so that its length is one.

首先对向量进行归一化，使其长度为1。

A similar vector is created for the query.

为查询创建了类似的向量。

The dot product of the query vector with each document vector is computed.

计算查询向量与每个文档向量的点积。

Vector Interpretation of the VSM
矢量解释VSM

$$match(d_j, q_k) = \frac{\sum_{i=1}^{n} (td_{ij} \times tq_{ik})}{\sqrt{\sum_{i=1}^{n} td_{ij}^2 \times \sum_{i=1}^{n} tq_{ik}^2}}$$

td_{ij}	TF*IDF weight on i^{th} term in the vector for document j	对于文档j的向量中的第i个术语的TF * IDF权重
tq_{ik}	TF*IDF weight on i^{th} term in the vector for query k	对于查询k，向量中的第i个项的TF * IDF权重
n	number of unique terms	唯一条款的数量

(cf. Manning & Schütze 6.3.2 Eqn 6.12)

OKAPI

Stephen E. Robertson 1946-Present

Stephen Robertson is a British computer scientist.

He is known for his work on information retrieval and the Okapi BM25 weighting model.

Wikipedia

OKAPI Term Weighting
OKAPI术语加权

The *Collection Frequency Weight* for a term is

文件收集频率权重

(1) CFW (i) = log N - log n

where

n = the number of documents term t(i) occurs in
N = the number of documents in the collection

包含术语t（i）的文件数量
文档集合中的文档数量

(cf. IDF definition above)

The *Term Frequency* for term t(i) in document d(j) is:

文件d（j）中术语t（i）的期限频率

TF (i,j) = the number of occurrences of term t(i) in document d(j)

文件d（j）中第t（i）项的出现次数

OKAPI Term Weighting
OKAPI术语加权

The Document Length of a document d(j) is:

文件的文件长度d（j）

DL (j) = the total number of term occurrences in document d(j)

文件d（j）中的术语总发生次数

The Normalised Document Length is:

规范化文档长度

NDL (j) = (DL (j)) / (Average DL for all documents)

OKAPI Term Weighting
OKAPI术语加权

For one term t(i) and one document d(j), the Combined Weight is 组合重量

$$CW(i,j) = \frac{[\,CFW(i) * TF(i,j) * (K1+1)\,]}{[\,K1 * (\,(1-b) + (b * (NDL(j))\,)\,) + TF(i,j)\,]}$$

(cf. Manning & Schütze 11.4.3 Eqn 11.33)

K1 influences the effect of term frequency.
A value of 2 gives good results.

K1影响术语频率的影响。
值为2会产生良好的结果。

b influences the effect of document length.
A value of 0.75 gives good results.

b影响文件长度的影响。
值0.75可以得到很好的结果。

Comparing TF*IDF and OKAPI
比较TF * IDF和OKAPI

CFW in OKAPI is like IDF in TF*IDF.

OKAPI中的CFW就像TF * IDF中的IDF一样。

TF and NDL within OKAPI are like hfreq without log flattening.

OKAPI中的TF和NDL就像没有日志展平的hfreq。

So, OKAPI is different from TF*IDF but not so different.

所以，OKAPI与TF * IDF不同，但没有那么不同。

Using OKAPI for Retrieval
使用OKAPI进行检索

To find the score for a document j against a query:

Compute the sum of CW(i,j) values for all terms i in the query

要根据查询查找文档j的分数：

计算查询中所有项i的CW（i，j）值的总和

The Inverted Index
信息检索倒置索引

Creating the Inverted Index
创建倒置索引

The amount of data to be stored is usually very large.

要存储的数据量通常非常大。

Thus, disk-based storage and retrieval must be used.

因此，必须使用基于磁盘的存储和检索。

We will discuss several different approaches.

我们将讨论几种不同的方法。

Information and Data Structures
信息和数据结构

Information to be stored:

- Term
- Documents in which it occurred
- Term weight (based on frequency in document)
- Exactly where it occurred (e.g. for proximity searches)

术语
发生的文件
术语重量（基于频率）
它发生的地方

Datastructures, three main types:

- Sorted array stored as a text file
- Tree structure stored as a set of text files
- Hash table

在文本文件中排序的数组
树结构存储了一组文本文件
哈希表

Constructing Sorted Array (cf. M&S 1.2)
构造排序数组

1. Go through docs
2. get stemmed terms
3. delete terms in stoplist
4. write term, file ID

浏览文档
得到词干
删除停止列表中的条款
写术语，文件ID

report 1 ('report' is term, '1' is file ID)
pap 1
report 1
...
report 2
human 2
...

Constructing Sorted Array cont.
构造排序数组

Sort alphabetically:

按字母顺序排序：

human 2
pap 1
report 1
report 1
report 2
...

Compute frequencies of terms in a file:

计算文件中术语的频率：

human 2 1 ('human'=term, '2'=File ID, '1'=Freq
pap 1 1
report 1 2
report 2 1

Searching the Sorted Array
在排序的数组中搜索

In simplest case, done using a standard binary search.

在最简单的情况下，使用标准二进制搜索完成。

Divide into index file and postings file:

分为“index”文件和“postings”文件：

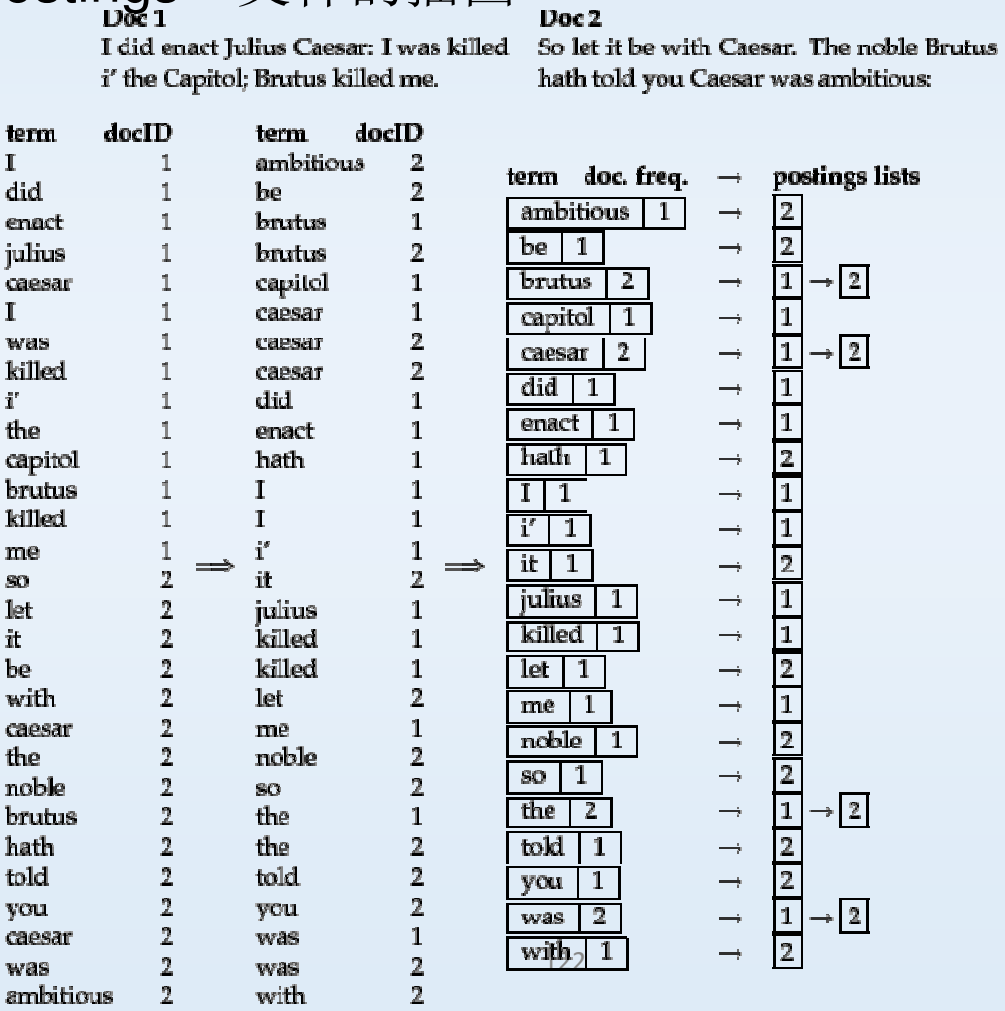
Index file:
Term
Number of postings
Offset in postings file

Postings File:
<File ID, Term Weight> pairs in fixed length fields

May have other information in postings file depending on the search algorithms supported.

根据支持的搜索算法，可能在发布文件中包含其他信息。

Simple Illustration (M&S 1.2 Fig. 1.4)
“index”和“postings”文件的插图



Explanation of Diagram
图解释

First two columns: Words in text order and their docids (numbers).

Second two columns: Sorted alphabetically by word.

Last two columns: Index ('dictionary' in M&S) and postings list.
- They compute number of docs containing each term.
- They do not compute frequency of term in each document.

Here, 'doc freq' means the number of docids which contain the term, not the frequency of the term!

doc freq is thus the number of postings for the term.

Each posting is just the docid, no frequency

前两列：文本中的单词，docids（数字）。
第二列：按字母顺序排序。

最后两列：索引，帖子
注意：包含每个术语的文档数量，而不是术语频率

'doc freq'=包含该术语的docids数，而不是该术语的频率！

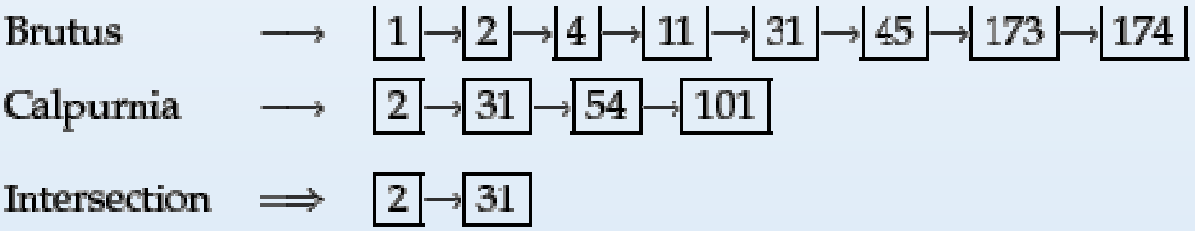
因此，doc freq是该术语的发布数量。
每个帖子只是docid，没有频率

AND Searching in a Boolean System
AND在布尔系统中搜索

Query: brutus AND calpurnia
(or it could just be: brutus calpurnia)

1. Locate Brutus in the index (dictionary)
2. Retrieve its postings
3. Locate Calpurnia in index
4. Retrieve its postings
5. Intersect the two postings lists

Efficiency: *linear* in the number of postings in both lists
Lists must be ordered by increasing docID.



1.在索引（字典）中找到Brutus
2.检索其发布内容
3.在索引中找到Calpurnia
4.检索其发布内容
5.将两个过帐列表相交

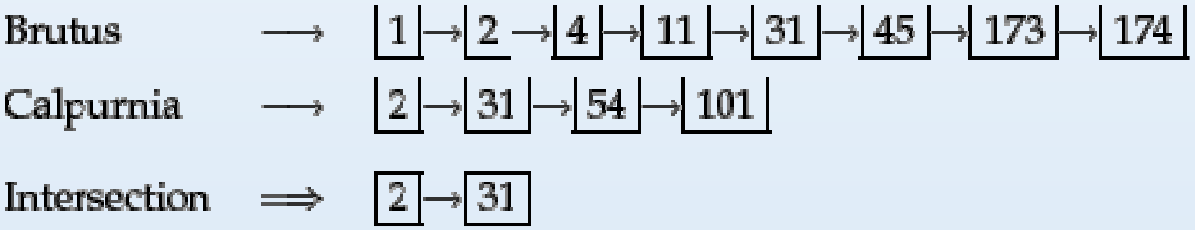
效率：两个列表中的过帐数量呈线性关系

必须订购清单

AND Searching in a Boolean System cont.
AND在布尔系统中搜索

Method of intersection:

1. Walk through the two postings lists simultaneously using two pointers.
2. Compare docID pointed to by both pointers. If same, put docID in results list, and advance both pointers.
3. Otherwise advance pointer pointing next to the smaller docID.
4. If at end of one of the lists, finish.



使用两个指针遍历两个列表。

比较两个指针指向的docID。如果相同，将docID放在结果列表中，并推进两个指针。

否则，将指针指向较小的docID。

如果在其中一个列表的末尾，则完成。

OR Searching in a Boolean System
或在布尔系统中搜索

Query: brutus OR calpurnia

1. Retrieve postings lists
2. Merge them, eliminating repeated docIDs

检索帖子列表
合并它们，消除重复的docID

OR Searching in a Boolean System cont.
或在布尔系统中搜索

Method of merging:

1. Walk through the two postings lists simultaneously using two pointers.

使用两个指针遍历两个列表。

2. Compare docID pointed to by both pointers. If not same put smaller docID in results list, and advance pointer to that smaller docID.

比较两个指针指向的docID。如果不相同，请在结果列表中放置较小的docID，将指针指向较小的docID。

3. If docIDs are same, put docID in results list and advance both pointers.

如果docID相同，请将docID放在结果列表中并推进两个指针。

4. If at end of one of the lists, add all remaining docIDs in remaining list to results.

如果在其中一个列表的末尾，则将剩余列表中的所有剩余docID添加到结果中。

AND NOT Query
“AND NOT”查询

Query: brutus AND NOT calpurnia

Similar to brutus AND Calpurnia:

1. Retain only the docIDs from the first list

仅保留第一个列表中的docID

2. Delete docIDs in the first list which are also in the second list

删除第一个列表中也位于第二个列表中的docID

AND NOT Query cont.
“AND NOT”查询

Method for List-1 AND NOT List-2:

浏览列表

1. Walk through the two postings lists simultaneously using two pointers.

比较两个指针指向的docID。

2. Compare docID pointed to by both pointers. If not same AND List-1 docID is smaller, put List-1 docID in results list and advance List-1 pointer.

如果不相同AND List-1 docID较小，则将List-1 docID放在结果列表中并前进List-1指针。

3. If not same AND List-2 docID is smaller, just advance List-2 pointer.

如果不相同AND List-2 docID较小，只需前进List-2指针。

4. If same, advance both List pointers.

如果相同，则前进两个List指针。

5. If at end of List-1, finish.

如果在List-1的末尾，则完成。

6. If at end of List-2 add all remaining docIDs in List-1 to results.

如果在List-2的末尾，将List-1中的所有剩余docID添加到结果中。

More Complex Boolean Queries
更复杂的布尔查询

Process in stages using the above methods.

使用上述方法分阶段处理。

Query: (brutus OR caesar) AND NOT calpurnia

Compute brutus OR caesar first

首先计算brutus或caesar

Then, do AND NOT calpurnia on resulting list

然后，在结果列表上执行AND NOT calpurnia

Query Processing Optimisation
查询处理优化

Query: brutus AND caesar AND calpurnia

To save time, we want to AND the shortest two lists first.

Assuming
brutus: 8 postings
caesar: more than 8 postings
calpurnia: 4 postings

We do calpurnia AND brutus first, then AND caesar second.

为了节省时间，我们首先想要最短的两个列表。
假设
brutus：8个贴子
凯撒：超过8个帖子
calpurnia：4个贴子

我们先做calpurnia和brutus，然后和凯撒第二次。

More General Optimisation
更多一般优化

(madding OR crowd) AND (ignoble OR strife)

Get number of postings for all terms.

Estimate size of each OR by adding the lengths of postings files
e.g. madding and crowd (linked by OR) vs. ignoble and strife (linked by OR).

Process in increasing order of OR sizes.

获取所有条款的过帐数量。

通过添加发布文件的长度来估算每个OR的大小，例如疯狂和人群（由OR联系）与卑鄙和冲突（由OR联系）。
按OR大小的顺序处理。

Skip Pointers (M&S 2.3)
“跳过”指针

Intersecting postings lists is fairly efficient (linear in the total length of both postings lists)

相交的过帐列表相当有效（两个过帐列表的总长度呈线性）

However, postings lists can be very long - millions of entries or even more

但是，帖子列表可能很长 - 数百万条目甚至更多

Skip pointers allow us to skip postings in one list which we know are missing from the other list

跳过指针允许我们跳过一个列表中的帖子，我们知道其他列表中缺少这些列表

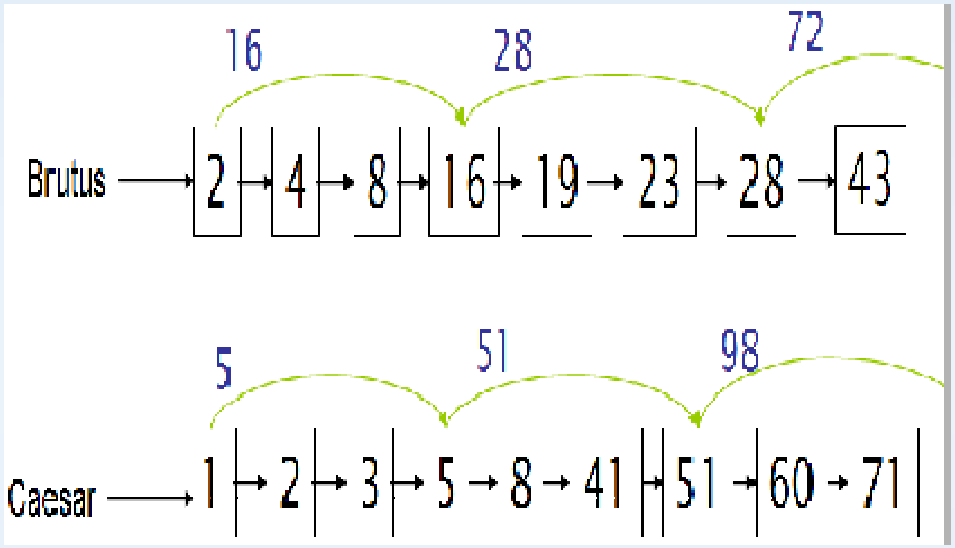
Where to put Skip pointers

在哪里放Skip指针？

How to make sure results do not change?

如何确保结果不会改变？

Skip Pointers
“跳过”指针

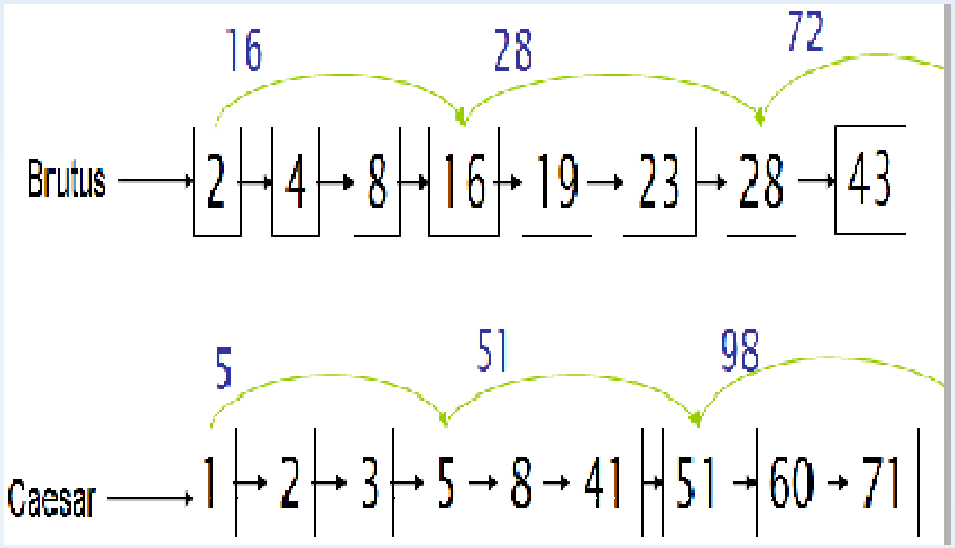


Skip pointer says value of docid which will be reached by skipping.

该跳过指针表示将通过跳过达到的docid值

e.g. At Brutus docid 16 skip takes us to 28

Skip Pointers
“跳过”指针



Example: We are at Brutus 8 and Caesar 8
Advance both pointers: Brutus 16 and Caesar 41
Normally, advance to Brutus 19

However, skip pointer goes to Brutus 28 -
still smaller than Caesar 41. So, go to Brutus 28
Caesar 41 etc

例如：我们在Brutus 8和Caesar 8
推进两个指针：Brutus 16和Caesar 41
通常情况下，前往布鲁图斯19

然而，跳过指针指向Brutus 28 - 仍然比
Caesar 41小。所以，去Brutus 28 Caesar 41

Where to Place Skip Pointers
在哪里放置Skip Pointers

Close together:
1. Used more
2. Take more time to check the pointers
3. Do not take you far
4. So saving in time is small

Far apart:
1. Used less
2. Take you further

紧靠在一起：
使用更多
花更多时间检查指针
不要带你走远
所以节省时间很少

离的远：
用得少
带你走吧

Where to Place Skip Pointers
在哪里放置Skip Pointers

Heuristic:

For postings list of length *P*, use *sqrt(P)* skips, equally spaced.

Note: Heuristic is a rule which *usually* works.

启发式：

对于长度为P的过帐列表，请使用等间距的sqrt（P）跳过。

注意：启发式是一种通常有效的规则。

Phrase Queries (M&S 2.4)
短语查询

10% of web queries are phrase queries!

10%的网络查询是短语查询！

e.g. “Stanford University”

例如“斯坦福大学”

Two approaches: Biword Indexes and Positional Indexes

两种方法：双字索引和位置索引

Phrase Queries cont.
短语查询

Biword Indexes

Divide query into pairs of words:

```
stanford university palo alto
=>
stanford university
university palo
palo alto
```

Note: They overlap

注意：它们重叠

Index collection with [stanford university] etc as single index terms appearing in postings etc.

与[stanford university]等的索引收集作为帖子中出现的单个索引术语等。

Problems:
1. False positives
2. Indexing overhead

问题：
1.误报
2.索引开销

Function Words in Biword Indexes
双字索引中的功能词

`university of essex`
`university at essex`
etc - how to match with function words?

In index include extended biwords:
- content words (university,essex) separated by
- function words (of,at)

Extract extended biwords from query:
query: `university at essex`
content words: `university`, `essex`
extended biword: [`university essex`]
=> get posting list for [`university essex`]

等 - 如何匹配功能词？

在索引中包含扩展的双字：
内容词（大学，艾塞克斯）分开
功能词（of，at）

从查询中提取扩展的双字：

Biword Indexes - Good and Bad Points
双字索引的好与坏点

Advantages:
- Can recognize many common two-word phrases
- Does not require the basic indexing mechanism to be changed

好处：
可以识别许多常见的双字短语
不需要更改基本索引机制

Disadvantages
- Multiple word phrases may be mis-matched

缺点
多个单词短语可能不匹配

`university of essex`
matches
`university in essex`

Positional Indexes (M&S 2.4.2)
位置指数

Record the position of each term in each doc:

记录每个文档中每个术语的位置：

```
docID, Freq_in_docID: <position1, position2, . . . >

to, 993427:
       <      1, 6: <7, 18, 33, 72, 86, 231>;
              2, 5: <1, 17, 74, 222, 255>;
              4, 5: <8, 16, 190, 429, 433>; ... >

be, 178239:
       <      1, 2: <17, 25>;
              4, 5: <17, 191, 291, 430, 434>;
              5, 3: <14, 19, 101>; ... >
```

'to' appears in 993,427 docs

'to'出现在993,427个文档中

It appears 6 times in doc 1 at positions 7, 18 etc.

它在doc 1的第7,18位等处出现了6次。

Example Search with Positional Index
如何使用位置索引进行搜索的示例

Search for ‘to be or not to be’:

Offset:	*n*	*n+1*	*n+2*	*n+3*	*n+4*	*n+5*
Word:	to	be	or	not	to	be

1. Obtain postings for ‘to’, ‘be’, ‘not’
2. Consider matching ‘to’ & ‘be’ as an example
3. Find postings which obtain both terms
4. Check positional criteria for a match:

Need to find:	AND in **SAME** doc:
‘to’ in posn *n*	‘to’ in posn *n+4*
‘be’ in posn *n+1* (i.e. adjacent)	‘be’ in posn *n+5*

DocID 4 satisfies search:

to	4, 5: <... 429, 433>; ... >
be	4, 5: <... 430, 434>;

1.获取'to'，'be'，'not'的帖子
2.考虑将'to'和'be'匹配为例
3.查找获得这两个术语的帖子
4.检查匹配的位置标准：

DocID 4满足搜索条件：

Proximity Search
邻近搜索

A similar approach can be taken for a search like

employment /3 place

i.e. 'place' within three words of 'employment'

1. Obtain postings for ‘place’, ‘employment’
2. Find docs which obtain both terms
3. Check positional criteria for a match

if 'employment' is in position *n*, 'place' needs to be in one of the positions *n-3, n-2, n-1, n+1, n+2, n+3*

可以采用类似的方法进行搜索

即'地方'在“就业”的三个字内

1.获取“地点”，“就业”的帖子
2.查找获得这两个术语的帖子
3.检查匹配的位置标准

如果'就业'在位置n，'地点'需要在n-3，n-2，n-1，n + 1，n + 2，n + 3的位置之一

Efficiency Implications of Positional Index
位置指数的效率

Postings intersection is slower:

- Now bounded by no. of tokens in doc collection
- No longer bounded just by no. of docs in doc collection.

Space taken is more:

- 2-4 times larger index than non-positional index
- When compressed, 1/3-1/2 size of text documents

查找列表的交集较慢：

所花费的时间与文档集合中的令牌数量成正比

占用的空间更多：

索引比非位置索引大2-4倍
压缩时，1 / 3-1 / 2大小的文本文件

Combining Biwords and Position Index
结合双字和位置索引

Biwords is much faster than postings intersection as no intersection needed.

Biwords比列表交集要快得多

So, Biwords are good for common terms
Posn index for rarer terms

因此，Biwords适用于常用术语
位置指数适用于罕见的术语

Analyse query logs to tell what is common

分析查询日志以告知什么是常见的

Also Biwords good where:

Biwords也很好：

Individual terms common
Desired phrase is rare

个别条款共同
搜索短语很少见

e.g. ...

Combining Biwords and Position Index
结合双字和位置索引

1. Britney Spears
- terms rare (SHORT postings)
- phrase common
so not much advantage in Biword

vs.

2. The Who
- terms common (LONG postings)
- phrase rare
so greater advantage to Biword

布兰妮斯皮尔斯
- 术语罕见（简短发布）
- 短语常见
所以Biword没有多大优势

与

2.谁
- 共同条款（LONG帖子）
- 短语罕见
Biword的优势更大

The Index (Dictionary, M&S 3.1)
信息检索指数

Index (Dictionary): Mapping from term to postings list

索引（词典）：从术语到发布列表的映射

Need to be able to

需要能够

Search it efficiently
Store it compactly
Update it easily

有效地搜索它
紧凑地存放
轻松更新

- conflicting aims!

- 相互冲突的目标！

Using Hashing for Index
使用Hashing函数进行信息检索索引

Allocate a lot of memory to Index

Convert input term to a number *n* somehow (clever bit)

Go to position *n* to find pointer to postings list

为Index分配大量内存

以某种方式将输入项转换为数字n（聪明位）

转到位置n以查找指向过帐列表的指针

Example Hashing Algorithm
简单的哈希算法示例

Add up the Ascii codes of the key and calculate the remainder after dividing by N (i.e. modulo)

加上密钥的Ascii代码并计算除以N后的余数（即模数）

Example (N=20):

Word	Ascii codes	Sum	Mod 20	Mod 5
lion	[108,105,111,110]	434	14	4
tiger	[116,105,103,101,114]	539	19	4
wolf	[119,111,108,102]	440	00	0
fox	[102,111,120]	333	13	3
rabbit	[114,97,98,98,105,116]	628	08	3

So, using Mod 20 column:

1. Create an Array called data
2. Store data for lion at data[14] etc.

因此，使用Mod 20列：
创建一个名为'data'的数组
将“lion”数据存储在data[14]等中。

Comments on Hashing Algorithm
关于哈希算法的评论

For a given N, the smallest hash will be 0 and the largest will be N-1 (because it is modulo i.e remainder N), i.e. N values in total.

最小的散列将为0，最大的散列将为N-1。

So we can scale our range of values by choosing a suitable N. e.g. we wish 100 values, use N=100.

我们可以通过选择N来确定我们的值范围。

Notice that for N=5 there were collisions - lion=tiger=4, fox=rabbit=3.

请注意，对于N = 5，存在碰撞 - lion = tiger = 4，fox = rabbit = 3。

A perfect hash function would never have collisions (does not exist).

完美的哈希函数永远不会发生冲突（不存在）。

Comments on Hashing Algorithm cont.
关于哈希算法的评论

In practical applications we can store values in a linked list in the case of collisions:

在实际应用中，我们可以在碰撞的情况下将值存储在链表中：

- e.g. look up tiger and go to data[4];

例如查老虎，转到数据[4];

- find it is already used, so create a linked list, one entry for lion, one entry for tiger;

发现它已经被使用了，所以创建一个链表，狮子的一个条目，老虎的一个条目;

- On subsequent lookup, if we reach data[4] we must test input to see if it was lion or tiger.

在随后的查找中，如果我们到达数据[4]，我们必须测试输入以查看它是狮子还是老虎。

It is a kludge, but not too bad if it is not too frequent.

它是一种kludge，但如果不是太频繁也不会太糟糕。

Avoiding Collisions in Hashing
避免哈希冲突

Make sure table is much bigger than the number of data items:

- e.g. 1.5 to 2 times bigger is normal;
- Notice for mod 5, collisions exist in example, for mod 20 no collisions;
- However, many table elements will then be empty, i.e. data storage is sparse (inefficient);

Or, use a better hashing algorithm.

Generally, hashing is good if
- Very fast access is required;
- There is plenty of spare storage space.

确保表格大于数据项目的数量：

例如正常值大1.5至2倍;
对于mod 5的注意，例子中存在碰撞，但是对于mod 20，没有碰撞;
但是，许多表元素将是空的，即数据存储是稀疏的（低效）;

或者，使用更好的散列算法。

通常，散列是好的，如果
需要非常快速的访问;
有足够的备用存储空间。

Using Binary Array Search for Index
使用二进制数组搜索的信息检索索引

- Arrange terms in alphabetical order (i.e. must be sorted)
- Store each dictionary element in fixed length fields (term plus pointer to postings)
- Can use simple binary search

按字母顺序排列术语（即必须排序）
将每个字典元素存储在固定长度字段中（术语加指向发布的指针）
可以使用简单的二分查找

Algorithm:

1. Make range start to end
2. Take middle point of range and test against search (input) term
3. Hence make range either start->middle or middle-> end

4. If range is one word, check if it is search term, if so finished else not found
5. Otherwise, continue from Step 2.

让范围从头到尾
取中间点范围并对搜索（输入）项进行测试
因此，使范围从开始 - >中间或中间 - >结束
如果范围是一个单词，检查它是否是搜索词，如果是，则完成其他未找到
否则，从步骤2继续。

Example of Binary Search
二进制搜索的示例

Find 'sickle'

1. aardvark
2. arm
3. boat
4. hand
5. huygens
6. jar
7. sickle
8. torch
9. vase
10. zygote

1. range: 1-10
2. mid point 6 'jar'
3. sickle > jar so new range 7-10
4. mid point 9 'vase'
5. sickle < vase so new range 7-8
6. mid point 8 'torch'
7. sickle < torch so new range 7-7
8. sickle **found**!

Advantage and Disadvantages of Binary Search
二进制搜索的优缺点

Advantages:

- Efficiency *(O) $Log_2(n)$* for *n* entries in index
- Easy to implement
- Suitable for in-memory index or disk-based index
- Can be very large

Disadvantages:

- Cannot add new entries
- Cannot delete old entries

好处：

索引中n个条目的效率（*O*）*Log_2*（*n*）

易于实施

适用于内存索引或基于磁盘的索引

可以非常大

缺点：

无法添加新条目

无法删除旧条目

Using Binary Tree Search for Index
使用二叉树搜索信息检索索引

- Similar to Binary Search
- Uses tree data structure not array
- Can still be on disk...

Have a series of structures containing:
1. token
2. pointer to 'less' words alphabetically
3. pointer to 'greater' words alphabetically
4. if 'leaf' node, pointer to postings

与二进制搜索类似
使用树数据结构而不是数组
仍然可以在磁盘上...

有一系列结构包含：
代币
按字母顺序指向'less'字样
按字母顺序指向“更大”的单词
如果是“叶子”节点，则指向帖子列表

Binary Tree Algorithms
二叉树算法

To Search Binary Tree:
1. Compare search token to current node
2. If less, go down 'less' pointer and go to Step 1
3. If greater, go down 'greater' pointer and go to Step 1
4. Otherwise, found!

To Add to Tree:
1. Search to leaf node (null 'less' & 'greater' pointers)
2. If new token is less, add new structure and make 'less' pointer point to it
3. If new token is geater, add new structure and make 'greater' pointer point to it

要搜索二叉树：
将搜索令牌与当前节点进行比较
如果更少，请按“少”指针并转到步骤1
如果更大，请按下“更大”指针并转到步骤1
否则，找到了！

要添加到树：
搜索叶节点（null'less'和'more'指针）
如果新标记较少，则添加新结构并使“less”指针指向它
如果新标记是geater，则添加新结构并使“更大”指针指向它

Binary Tree Algorithms cont.
二叉树算法

To Create Tree at the Start:
1. Make a node for first token
2. Add all other tokens as above

要在开始时创建树：
为第一个令牌创建一个节点
如上所述添加所有其他令牌

Binary Tree Advantages and Disadvantages
二叉树的优点和缺点

Advantages:
1. Easy to implement
2. Efficient - *(O) $Log_2(n)$* for *n* entries in index as before

3. Can add new entries
4. Can be in memory or disk-based

Disadvantage:
1. Uses more space than simple array

好处：
易于实施
高效 -（O）Log2（n）用于索引中的n个条目，如前所述
可以添加新条目
可以在内存或基于磁盘

坏处：
比简单数组使用更多空间

Binary Tree Properties
二叉树属性

For best results, tree must be **Balanced**:

> Number of nodes under the left and right pointers of any node should be equal or differ by one.

Also initial data used to create tree must be **random** not sorted - otherwise you get a linked list (slow)!

Adding and deleting elements can result in tree becoming *less balanced*

There are algorithms for re-balancing though this is costly especially if tree is on disk.

为获得最佳结果，树必须是平衡的：

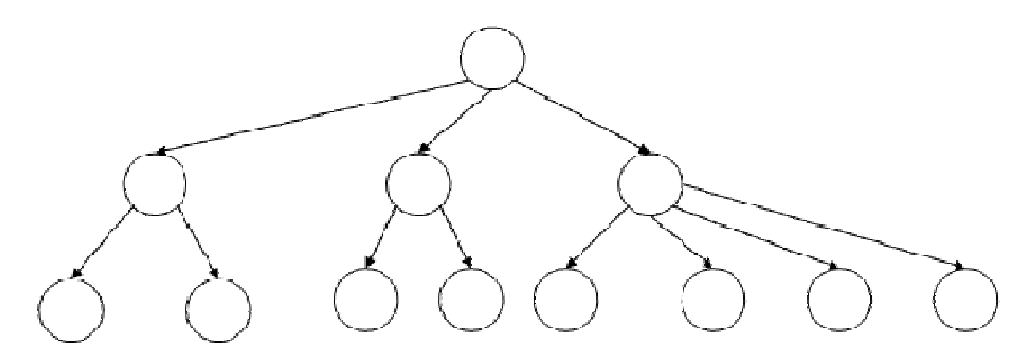
任何节点左右指针下的节点数应相等或相差一。

用于创建树的初始数据也必须是随机的而不是排序的 - 否则你会得到一个链表（慢）！

添加和删除元素可能导致树变得不那么平衡

有重新平衡的算法虽然这是昂贵的，特别是如果树在磁盘上。

B-Tree
B树



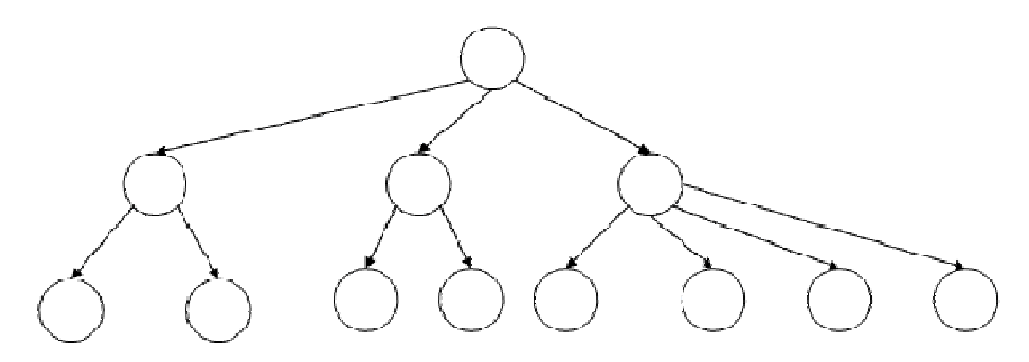
B-Tree: Every node has between *a* and *b* successors (e.g. *a=2*, *b=4* in figure above).

Search is similar except that every node with n successors needs n-1 comparison keys:

S1 k1 S2 k2 S3 k3 ... Sn

B树：每个节点都有a和b后继节点（例如上图中a = 2，b = 4）。
搜索类似，只是具有n个后继的每个节点都需要n-1个比较键：

B-Tree cont.
期



Assuming we are searching for key k and it is not at current node:

1. if k < k1, go down S1
2. if k1 <= k < k2, go down S2
3. if k2 <= k < k3, go down S3
4. ...
5. if k >= kn-1, go down Sn

This tree is broader and shallower than a binary tree:

1. Disk access (of a node) is less frequent
2. More comparisons are done at each node

这棵树比二叉树更宽更浅：
（节点的）磁盘访问频率较低
在每个节点进行更多比较

Advantages of B-Trees for IR
B树的优点用于信息检索

If stored on disk, number of retrievals will be less as tree is shallower.

如果存储在磁盘上，由于树较浅，检索次数会减少。

Computation at each node is more, but this is done by CPU.

每个节点的计算更多，但这是由CPU完成的。

Wildcard Queries (M&S 3.2)
通配符查询

User may wish to specify search term partially:

e.g. `mon*` matches `monday`, `mono` etc

A search tree index of the IR dictionary can handle this:

- Assuming 26 letters, root has 26 successors, one for each.
- Each successor node has 26 successors, one for each letter

etc

用户可能希望部分指定搜索词：

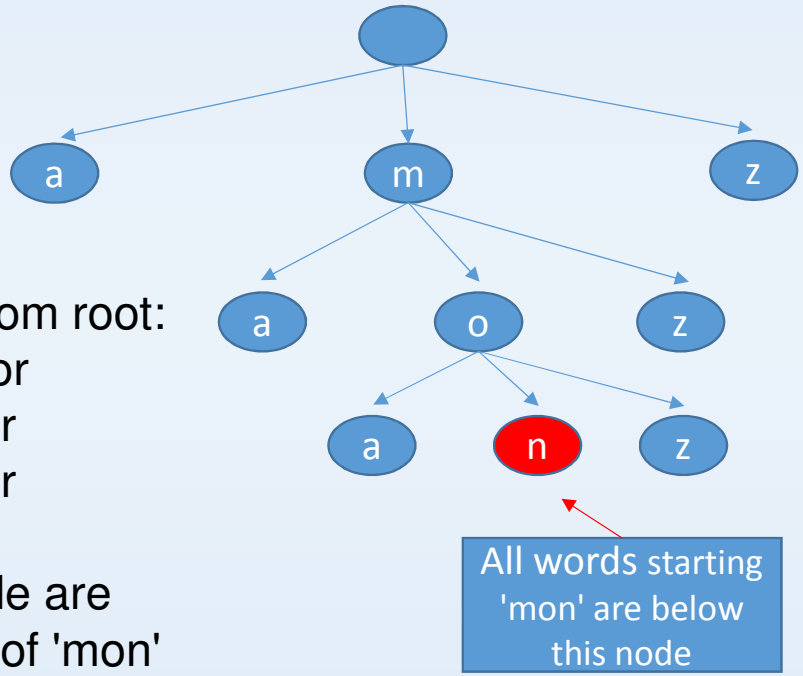
例如mon *匹配星期一，单声道等

IR字典的搜索树索引可以处理：

假设26个字母，root有26个继承者，每个有一个。
每个后继节点有26个后继，每个字母一个

等等

Wildcard Queries cont.
通配符查询



Searching for mon* from root:
1. Select ‘m’ successor
2. Select ‘o’ successor
3. Select ‘n’ successor

All children of red node are possible completions of 'mon'

However, only for **trailing wildcards** (at end).

所有红色节点的孩子都是'mon'可能的完成

但是，仅用于尾随通配符（结尾）。

Permuterm Indexes for Wildcard Queries
通配符查询的Permuterm索引

Allows general wildcard queries e.g.
`fi*mo*er` matches `fishmonger` etc

Permuterm vocabulary:
- Take word
- Add $
- Create rotations
- Link them all to original word, allowing to search on trailing wildcard as above

允许通用通配符查询，例如“fi * mo * er”fishmonger“鱼贩”等

Permuterm词汇：
从这个词开始
加$
创建旋转
将它们全部链接到原始单词，允许在上面搜索尾随通配符

Permuterm Index Example
Permuterm指数示例

hello$	man$	moron$	fishmonger$
->	->	->	->
ello$h	an$m	oron$m	ishmonger$f
llo$he	n$ma	ron$mo	shmonger$fi
lo$hel	$man	on$mor	hmonger$fis
o$hell		n$moro	monger$fish
$hello		$moron	onger$fishm
			nger$fishmo
			ger$fishmon
			er$fishmong
			r$fishmonge
			$fishmonger

Retrieval using Permuterm
使用Permuterm检索

for `X` lookup on `X$`
for `X*` lookup on `$X*`
for `*X` lookup on `X$*`
for `*X*` lookup on `X*`
for `X*Y` lookup on `Y$X*`

e.g. `mo*` lookup on `$mo*`
- matches `$moron -> moron`

e.g `m*n` lookup on `n$m*`
- matches `n$ma -> man`, `n$moro -> moron`

用于在X $上进行X查找
etc.

Permuterm with Two Wildcards
Permuterm与两个通配符

How to find `fi*mo*er` ?

Two steps:

1. Treat as `fi*er` and search for `er$fi*`
2. Check all matches and select those containing `mo`

e.g. `fishmonger` will match.

如何找到“fi * mo * er”?

两个步骤：

1.视为“fi * er”并搜索“er $ fi *”

2.检查所有匹配并选择包含mo的匹配
例如“鱼贩”将匹配。

k-gram Indexes for Wildcard Queries
通配符查询的k-gram索引

k-gram: Seq of *k* characters; suppose *k=3*

k-gram：k个字符的seq;假设k = 3

e.g. `metric` (make it `$metric$`)

例如指标（使其成为$ metric $）

```
$me met etr tri ric ic$
```

Postings list goes from k-gram to words containing k-gram:

发布列表从k-gram到包含k-gram的单词：

```
etr -> beetroot -> metric -> petrify -> retrieval
```

Retrieval on k-Gram Index
检索k-Gram指数

1. Split query `s1*s2` into `$s1, s2$`
2. Convert each `$s1` and `s2$` into k-grams
3. Search k-gram index for the k-grams
4. Compute intersection of postings lists
5. Filter results to check they match `s1*s2`

将查询s1 * s2拆分为$ s1，s2 $
将每个$ s1和s2 $转换为k-gram
搜索k-gram的k-gram指数
计算发布列表的交集
过滤结果以检查它们是否匹配s1 * s2

Example of Retrieval on k-Gram Index
k-Gram索引的检索示例

`red*`
->
`$red` (no $ at end as it does not end there)
->
`$re, red` (k-grams for `$red`)

postings list for `$re`
->
`reduce, red, retired`

postings list for `red`
->
`reduce, red`

intersect postings lists
->
`reduce, red` (`retired` filtered out)

没有$结束，因为它没有结束）
（k-gram为$ red）

“$ re”的“postings”列表

“red”的“postings”列表

交叉“postings”列表
（“retired”过滤掉了）

K-gram Index vs. Permuterm Index
K-gram指数与Permuterm指数比较

k-gram is more space efficient

k-gram更节省空间

Permuterm does not require post filtering (in one-* case)

Permuterm不需要后期过滤（在一个*的情况下）