University of Southern California

USC Viterbi
School of Engineering

Inverted Indexing

Outline

- Definition of an Inverted index
- Examples of Inverted Indices
- Representing an Inverted Index
- Processing a Query on a Linked Inverted Index
- Skip Pointers to Improve Merging
- Phrase Queries
- biwords
- Grammatical Tagging
- N-Grams
- Distributed Indexing

USC Viterbi School of Engineering

Creating an Inverted Index

- An inverted index is typically composed of a vector containing all distinct words of the text collection in lexicographical order (which is called the **vocabulary**) and for each word in the vocabulary, a list of all documents (and text positions) in which that word occurs
  - This is nothing more than an index that one finds at the back of a book
- Terms in the inverted file index ***may be*** refined:
  - ***Case folding***: converting all letters either to upper case or to lower case
  - ***Stemming/lemmatization***: reducing words to their morphological roots
  - ***Stop words***: removing words that are so common they provide no information

Processing a Query An Example

- The Query
  - Which plays of Shakespeare contain the words *Brutus AND Caesar* but *NOT Calpurnia*?
- One Possible Solution
  - One could grep all of Shakespeare's plays for *Brutus* and *Caesar,* then strip out lines containing *Calpurnia*?
    - Too Slow (for large corpora)
    - Requires lots of space
    - This method doesn't allow for other operations (e.g., find the word *Romans* near *countrymen*) are not feasible with this approach

grep is a command-line utility for searching plain-text data sets for lines that match a regular expression.

USC Viterbi School of Engineering

Term-Document Incidence Matrix

One way to think about an inverted index is to consider it as a sparse matrix where rows represent terms and columns represent documents

terms ↗ / documents →	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

1 if the play contains word, 0 otherwise

The Query:
***Brutus** AND **Caesar** but NOT*
Calpurnia

Incidence Vectors

- So we have a 0/1 vector for each term.
- To answer the previous query: take the vectors for *Brutus, Caesar* and *Calpurnia* (complemented) and do a bitwise *AND*.
- 110100 *AND* 110111 *AND* 101111 = 100100
- So the two plays matching the query are:

  "Anthony and Cleopatra", "Hamlet"

Actual Answers to the Query

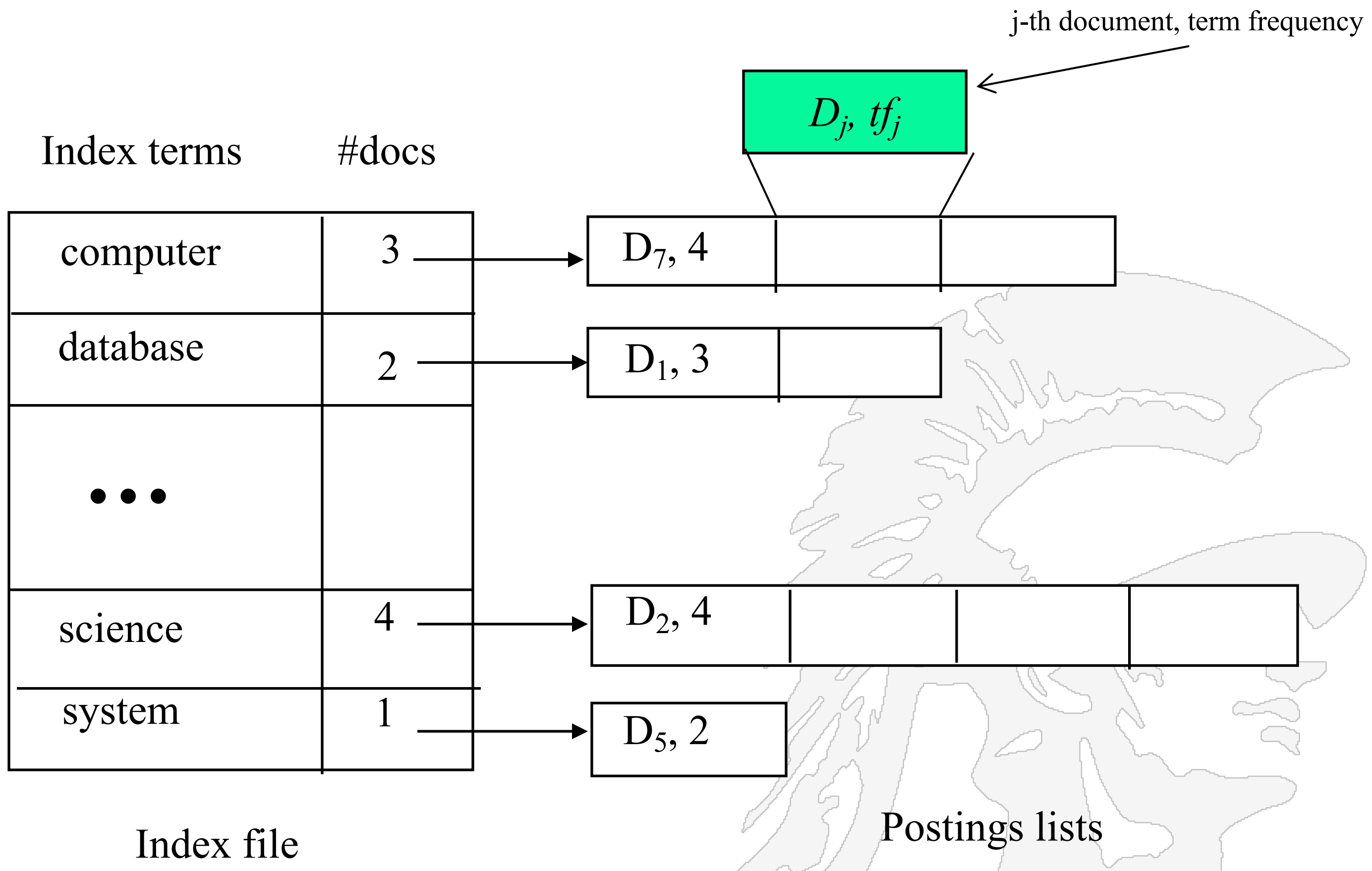
- **Antony and Cleopatra, Act III, Scene ii**
- ***Agrippa* [Aside to DOMITIUS ENOBARBUS]:**
- Why, Enobarbus,
- When Antony found Julius *Caesar* dead,
- He cried almost to roaring; and he wept
- When at Philippi he found *Brutus* slain.

- **Hamlet, Act III, Scene ii**
- *Lord Polonius:* I did enact Julius *Caesar* I was killed i' the Capitol; *Brutus* killed me.

Term-Document Incident Matrices are Naturally Sparse

- Given 1 million documents and 500,000 terms
- The (term x document) matrix in this case will have size 500K x 1M or half-a-trillion 0's and 1's.
- But it will likely have no more than one billion 1's.
  - So the matrix is extremely sparse, only 0.1% of the elements are 1
- So instead we use a data structure for an inverted index that exploits sparsity and then devise algorithms for query processing

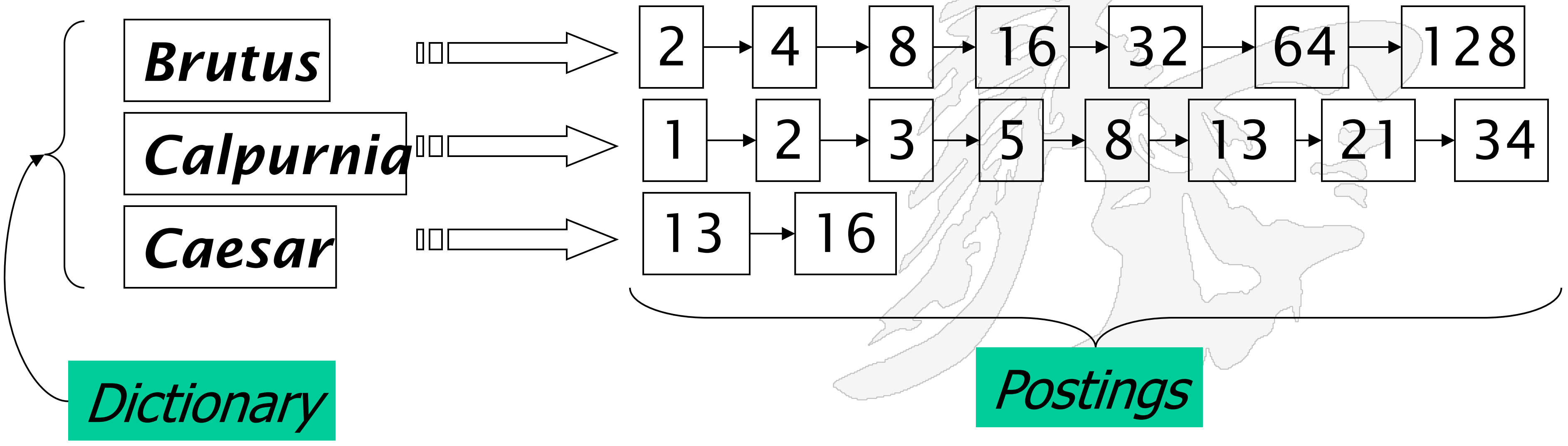
Inverted Index Example



The two parts of an inverted index. The **dictionary** (Index file) is usually kept in memory, with pointers to each **postings list** , which is stored on disk. The dictionary has been sorted alphabetically and the postings list is sorted by document ID

Inverted Index Stored In Two Parts

- For each term *T*, we must store a list of all documents that contain *T*.
- Linked lists are generally preferred to arrays, why . . .
  - Dynamic space allocation
  - Insertion of terms into documents easy
  - There is space overhead of pointers, though this is not too serious



Parsing Documents To Create an Inverted Index

- Documents are parsed to extract words and these are saved with the document ID i.e a sequence of (possibly modified token, Document ID) pairs

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

- If the corpus is known in advance, then after all documents have been parsed the inverted file is sorted by terms

Initial capture of terms →

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Refined list of terms →

Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

- However, on the Web, documents are constantly being added and the terms are constantly increasing

- Multiple term entries in a single document are merged.
- Frequency information is added.

Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

$\rightarrow$

Term	Doc #	Freq
ambitious	2	1
be	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
did	1	1
enact	1	1
hath	2	1
I	1	2
i'	1	1
it	2	1
julius	1	1
killed	1	2
let	2	1
me	1	1
noble	2	1
so	2	1
the	1	1
the	2	1
told	2	1
you	2	1
was	1	1
was	2	1
with	2	1

e.g. Caesar Occurs in Documents 1 and 2, With Total Frequency 3

- The file is commonly split into a *Dictionary* and a *Postings* file

Term	Doc #	Freq
ambitious	2	1
be	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
did	1	1
enact	1	1
hath	2	1
I	1	2
i'	1	1
it	2	1
julius	1	1
killed	1	2
let	2	1
me	1	1
noble	2	1
so	2	1
the	1	1
the	2	1
told	2	1
you	2	1
was	1	1
was	2	1
with	2	1

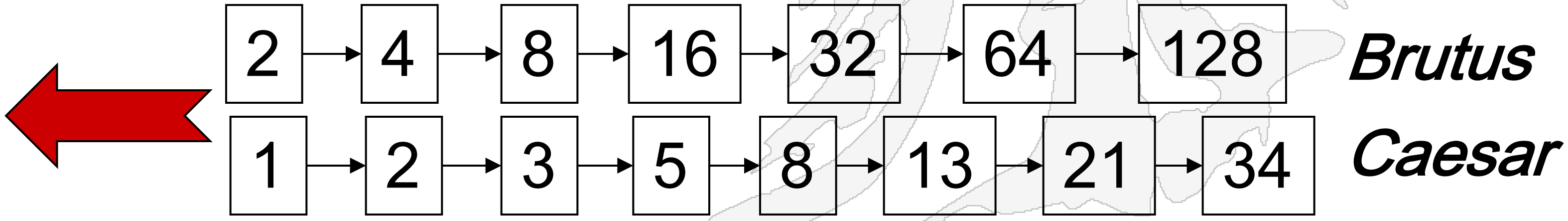
Term	N docs	Tot Freq
ambitious	1	1
be	1	1
brutus	2	2
capitol	1	1
caesar	2	3
did	1	1
enact	1	1
hath	1	1
I	1	2
i'	1	1
it	1	1
julius	1	1
killed	1	2
let	1	1
me	1	1
noble	1	1
so	1	1
the	2	2
told	1	1
you	1	1
was	2	2
with	1	1

Doc #	Freq
2	1
2	1
1	1
2	1
1	1
1	1
2	2
1	1
1	1
2	1
1	2
1	1
2	1
1	1
1	2
2	1
1	1
2	1
2	1
1	1
2	1
2	1
2	1
1	1
2	1
2	1

Query Processing Across the Postings List

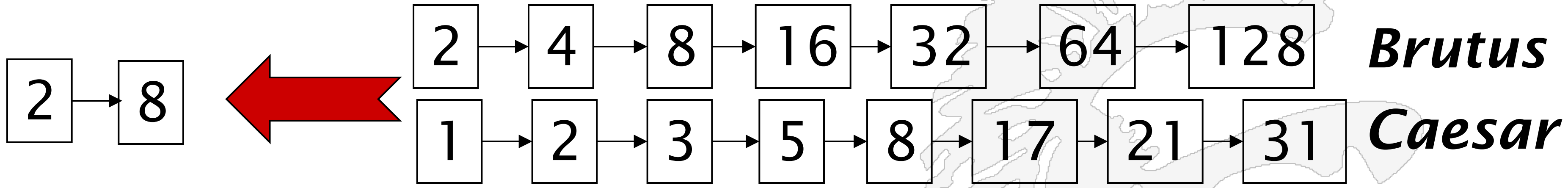
- **Consider processing the query:**

  Brutus AND Caesar
  - Locate *Brutus* in the Dictionary;
    - Retrieve its postings.
  - Locate *Caesar* in the Dictionary;
    - Retrieve its postings.
  - "Merge" the two postings and select the ones in common (postings are document ids):



Basic Merge

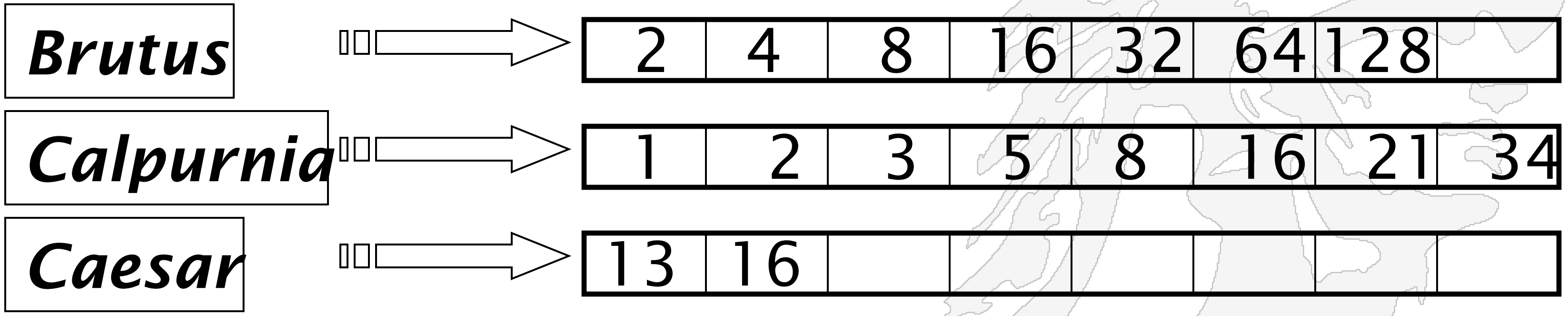
- Walk through the two postings simultaneously, in time linear in the total number of postings entries



If the list lengths are *m* and *n*, the merge takes O(*m*+*n*) operations.

Query Optimization

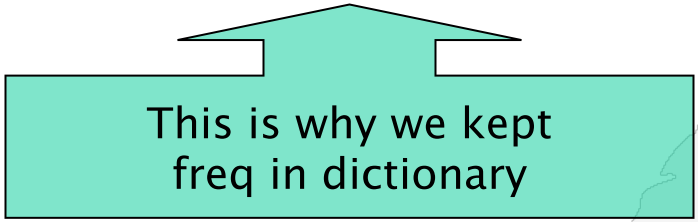
- What is the best order for query processing?
- Consider a query that is an *AND* of *t* terms.
- For each of the *t* terms, get its postings, then *AND* together.

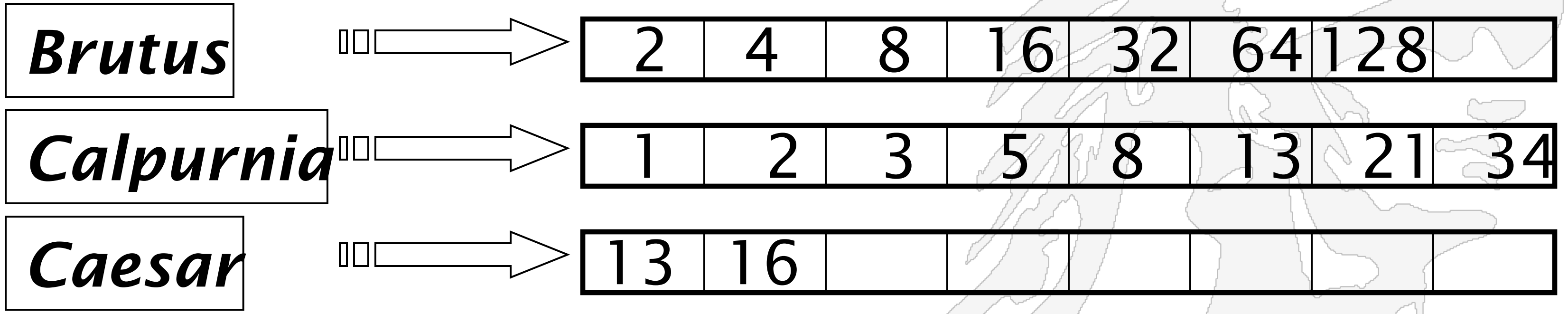


Query: ***Brutus*** *AND* ***Calpurnia*** *AND* ***Caesar***

Query Optimization Example

- Process in order of increasing frequency of occurrence:
  - *start with smallest set, then keep cutting further.*





Brutus	2	4	8	16	32	64	128	
Calpurnia	1	2	3	5	8	13	21	34
Caesar	13	16						

Execute the query as (***Caesar*** *AND* ***Brutus***) *AND* ***Calpurnia***.

Algorithm for the intersection of two postings lists p1 and p2

```
INTERSECT( p1, p2)
1   answer ← ( )
2   while p1 /= NIL and p2 /= NIL
3   do if docID( p1) = docID( p2)
4           then ADD(answer, docID( p1))
5                p1 ← next( p1)
6                p2 ← next( p2)
7           else  if docID( p1) < docID( p2)
8                   then p1 ← next( p1)
9                   else  p2 ← next( p2)
10  return answer
```

advance pointer on the smaller docID

Algorithm for conjunctive queries that returns the set of documents containing each term in the input list of terms

```
INTERSECT(<t_1, . . . , t_n>)
1   terms ← SORTBYINCREASINGFREQUENCY(<t_1, . . . , t_n>)
2   result ← postings( first(terms))
3   terms ← rest(terms)
4   while terms /= NIL and result /= NIL
5   do result ← INTERSECT(result, postings( first(terms)))
6       terms ← rest(terms)
7   return result
```

Algorithm Strategy:
intersect each retrieved postings list with the current intermediate result in memory, where we initialize the intermediate result by loading the postings list of the least frequent term

To speed up the merging of postings we use the technique of ***Skip Pointers***

The Technique of Skip Pointers

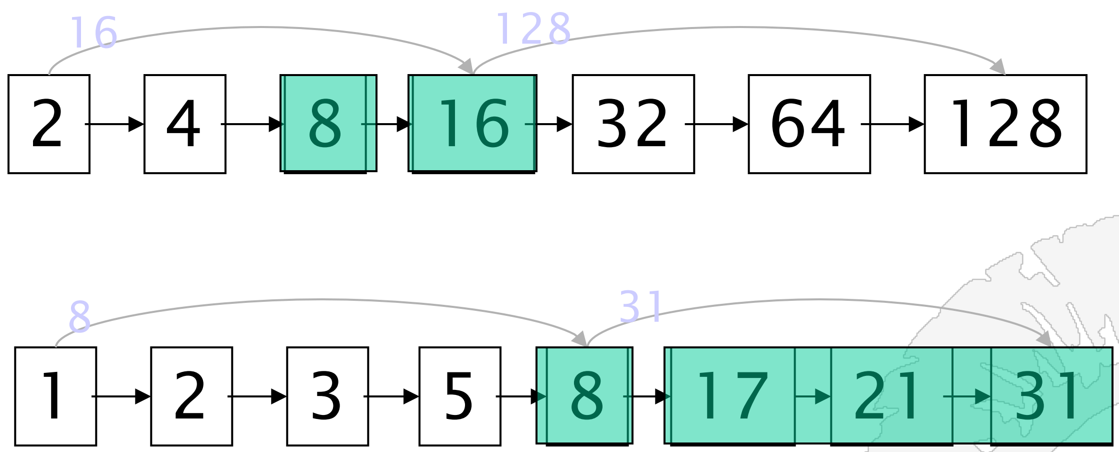
Augment postings with skip pointers (at indexing time)

$$\begin{tikzcd}
\boxed{2} \arrow[r] \arrow[rrr, bend left, "16"] & \boxed{4} \arrow[r] & \boxed{8} \arrow[r] & \boxed{16} \arrow[r] \arrow[rrr, bend left, "128"] & \boxed{32} \arrow[r] & \boxed{64} \arrow[r] & \boxed{128}
\end{tikzcd}$$

$$\begin{tikzcd}
\boxed{1} \arrow[r] \arrow[rrrr, bend left, "8"] & \boxed{2} \arrow[r] & \boxed{3} \arrow[r] & \boxed{5} \arrow[r] & \boxed{8} \arrow[r] \arrow[rrr, bend left, "31"] & \boxed{17} \arrow[r] & \boxed{21} \arrow[r] & \boxed{31}
\end{tikzcd}$$

- Why?
- <u>To skip postings that will not figure in the search results.</u>
- How?
- Where do we place skip pointers?

Query Processing With Skip Pointers



Suppose we've stepped through the lists until we process **8** on each list.

When we get to **16** on the top list, we see that its successor is **32**.

But the skip successor of **8** on the lower list is **31**, so we can skip ahead past the intervening postings 17 and 2

Facts on Skip Pointers

- **Skip pointers are added at indexing time**; they are shortcuts, and they only help for AND queries and they are useful when the corpus is relatively static
- there are two questions that must be answered:
- 1. where should they be placed?
- 2. how do the algorithms change?

- **The Argument:** More skips means shorter skip spans, and that we are more likely to skip. But it also means lots of comparisons to skip pointers, and lots of space storing skip pointers. Fewer skips means few pointer comparisons, but then long skip spans which means that there will be fewer opportunities to skip.

- **The Solution**: A simple heuristic for placing skips, which has been found to work well in practice, is that ***for a postings list of length P, use sqrt{P} evenly-spaced skip pointers.*** This heuristic possibly can be improved upon as it ignores any details of the distribution of query terms. **[Moffat and Zobel 1996]**

.

- See the YouTube video http://www.youtube.com/watch?v=tPsCQOsa7j0 (15 min)

Faster Postings List Intersection via Skip Pointers

```
INTERSECTWITHSKIPS($p_1, p_2$)
answer ← ⟨ ⟩
while $p_1 \neq$ NIL and $p_2 \neq$ NIL
do if $docID(p_1) = docID(p_2)$
      then ADD($answer, docID(p_1)$)
           $p_1 \leftarrow next(p_1)$
           $p_2 \leftarrow next(p_2)$
      else if $docID(p_1) < docID(p_2)$
              then if $hasSkip(p_1)$ and $(docID(skip(p_1)) \leq docID(p_2))$
                      then while $hasSkip(p_1)$ and $(docID(skip(p_1)) \leq docID(p_2))$
                              do $p_1 \leftarrow skip(p_1)$
                      else $p_1 \leftarrow next(p_1)$
              else if $hasSkip(p_2)$ and $(docID(skip(p_2)) \leq docID(p_1))$
                      then while $hasSkip(p_2)$ and $(docID(skip(p_2)) \leq docID(p_1))$
                              do $p_2 \leftarrow skip(p_2)$
                      else $p_2 \leftarrow next(p_2)$
return answer
```

- Skip pointers will only be available for the original postings lists.
- For an intermediate result in a complex query, the call hasSkip(p) will always return false.
- Finally, note that the presence of skip pointers only helps for AND queries, not for OR queries.

Phrase Queries

Phrase queries

- We want to answer queries such as *stanford university* – as a phrase
- Many search engines allow users to specify a phrase using double quotes, "Stanford University", which most people find easy to understand
- As many as 10% of web search engine queries are phrase queries and many more are implied phrase queries, e.g. George Bush, Bernie Sanders
- If we only store terms and documents in our inverted index, <*term* : *docs*> then how we can answer a phrase query

- There are two approaches to solving this problem

1. Bi-word indexes (also called 2-grams or 2 shingles)
2. Positional indexes

Using Biword Indexes for Phrase Searching

- Definition: A *bi-word* (or a 2-gram) is a consecutive pair of terms in some text
- To improve phrase searching one approach is to index every bi-word in the text
- For example the text "Friends, Romans, Countrymen" would generate the bi-words
  - *friends romans*
  - *romans countrymen*
- Each of these bi-words is now added to the dictionary as a term

Handling Longer Phrase Queries

- Consequences
  - Bi-words will cause an explosion in the vocabulary database
  - Queries longer than 2 words will have to be broken into bi-word segments
- Example: suppose the query is the 4 word phrase

stanford university palo alto

The query can be broken into the Boolean query on bi-words:

*stanford university **AND** university palo **AND** palo alto*

- Matching the query to terms in the index will work, but may also produce false positives (i.e. occurrences of the bi-words, but not the full 4 word query)
- A Bi-word index that is extended to longer sequences, or even variable length sequences is called a *phrase index*

Part-of-Speech Tagging

- **Many two word phrases have embedded stop words, e.g**
  - the abolition of slavery
  - negotiation of the constitution
- **To salvage the bi-word indexing method on these examples one can use part-of-speech tagging**
  - Part-of-speech taggers classify words as nouns, verbs, etc. – or, in practice, often as finer grained classes like “plural proper noun”.
  - “Negotiation of the constitution” is transformed into “N X X N”
- **Many fairly accurate (c. 96% per-tag accuracy) part-of-speech taggers now exist, usually trained by machine learning methods on hand-tagged text**
  - See Manning and Schutze “*Foundations of Statistical Natural Language Processing*”
  - https://nlp.stanford.edu/fsnlp/
  - https://www.issco.unige.ch/en/staff/robert/tatoo/tatoo.html

Alternate Solution - Using Positional Indexes

- Given the limitations of a bi-word index, (i.e. the enormous growth in the vocabulary) the alternate solution is most commonly used, called a **Positional Index**
- Store, for each *term* in the index, entries of the form:

  <term, number of docs containing *term*;

  ***doc1*: position1, position2, position3 … ;**

  ***doc2*: position1, position2 … ;**

  etc.>
- i.e. for each occurrence of the term in a document its position is stored

Positional Index Example

for each term in the vocabulary, we store postings of the form
docID: position1, position2, ...,
where each position is a token index in the document.
Each posting will also usually record the term frequency
Adopting a positional index expands required postings storage significantly, even if we compress position values/offsets

Lots of documents

Lots of occurrences

<*be*: 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, …>

- the term is "be", typically a stop word
- it occurs in 993,427 documents
- in document 1 "be" occurs at positions: 7, 18, 33, 72, 86 and 231

- **this scheme expands postings storage *substantially (rather than the vocabulary)***

Processing a Phrase Query

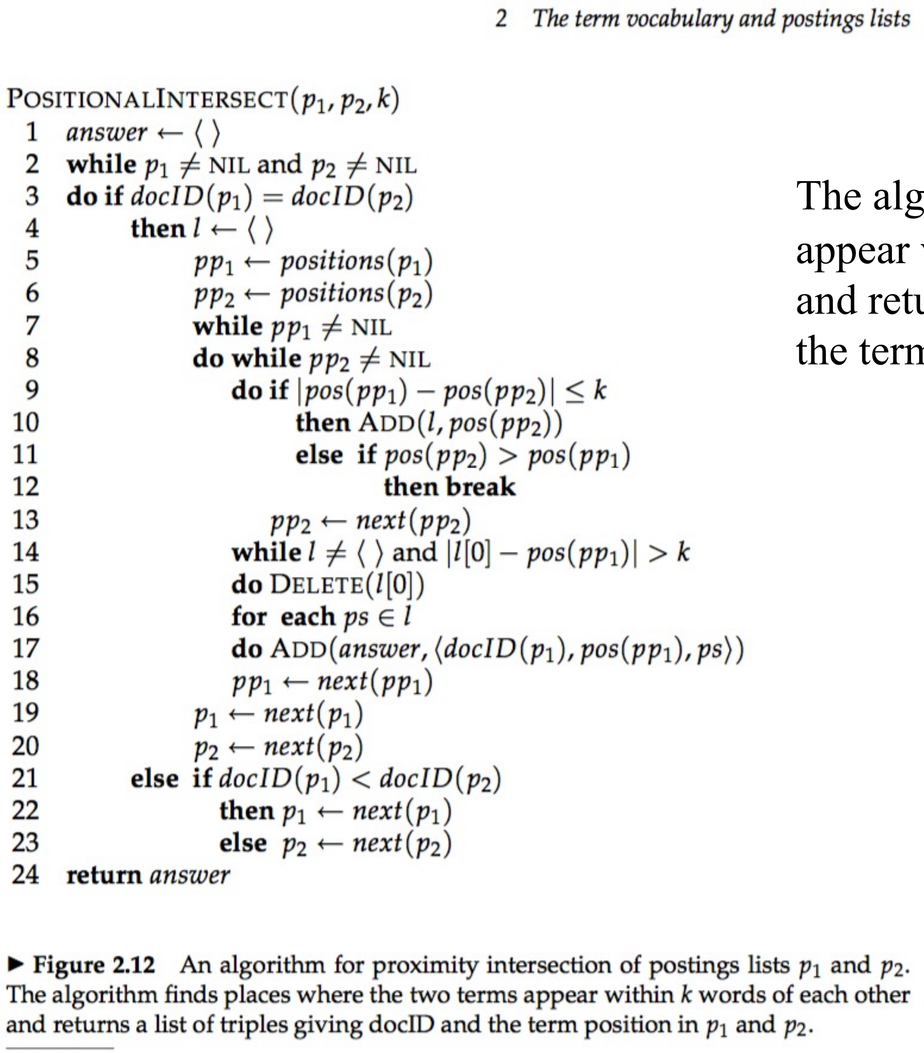
- Algorithm for matching a phrase query:
  1. **Extract** inverted index entries for each distinct term: e.g. *to, be, or, not, to, be*
  2. **Merge** their *doc:position* lists to enumerate all positions with "*to be or not to be*".
  3. **Match** those documents that contain the terms in the adjacent positions

  – ***to:***
    - ***2*:1,17,74,222,551; *4*:8,16,190,429,433; *7*:13,23,191; ...**

  – ***be:***
    - ***1*:17,19; *4*:17,191,291,430,434; *5*:14,19,101; ...**
    - **Same general method for proximity searches**

  – In document 4 the word "to" appears in position 16 and the word "be" appears in position 17, so they are adjacent

Algorithm for Proximity Queries with *k* words



```
PositionalIntersect($p_1, p_2, k$)
answer ← ⟨ ⟩
while $p_1 \neq$ NIL and $p_2 \neq$ NIL
do if $docID(p_1) = docID(p_2)$
     then $l$ ← ⟨ ⟩
          $pp_1 \leftarrow positions(p_1)$
          $pp_2 \leftarrow positions(p_2)$
          while $pp_1 \neq$ NIL
          do while $pp_2 \neq$ NIL
             do if $|pos(pp_1) - pos(pp_2)| \leq k$
                   then Add($l, pos(pp_2)$)
                   else if $pos(pp_2) > pos(pp_1)$
                           then break
                $pp_2 \leftarrow next(pp_2)$
             while $l \neq$ ⟨ ⟩ and $|l[0] - pos(pp_1)| > k$
             do Delete($l[0]$)
             for each $ps \in l$
             do Add(answer, ⟨$docID(p_1), pos(pp_1), ps$⟩)
             $pp_1 \leftarrow next(pp_1)$
          $p_1 \leftarrow next(p_1)$
          $p_2 \leftarrow next(p_2)$
     else if $docID(p_1) < docID(p_2)$
             then $p_1 \leftarrow next(p_1)$
             else $p_2 \leftarrow next(p_2)$
return answer
```

► **Figure 2.12** An algorithm for proximity intersection of postings lists p_1 and p_2. The algorithm finds places where the two terms appear within k words of each other and returns a list of triples giving docID and the term position in p_1 and p_2.

The algorithm finds places where the two terms appear within *k* words of each other and returns a list of triples giving docID and the term position in p_1 and p_2.

Some High Freqency Noun Phrases from TREC and Patent DataSets

TREC		*Patent*	
Frequency	***Phrase***	***Frequency***	***Phrase***
65824	**United States**	**975362**	**present invention**
61327	**article type**	**191625**	**u.s. pat**
33864	**Los Angeles**	**147352**	**preferred embodiment**
18062	**Hong Kong**	**95097**	**carbon atoms**
17788	**North Korea**	**87903**	**group consisting**
17308	**New York**	**81809**	**room temperature**
15513	**San Diego**	**78458**	**seq id**
15009	**Orange county**	**75850**	**brief description**

The phrases above were identified by POS tagging; The data above shows that common phrases are used more frequently in patent data as patents have a very formal style; many of the TREC phrases are proper nouns, whereas patent phrases are those that occur in all patents

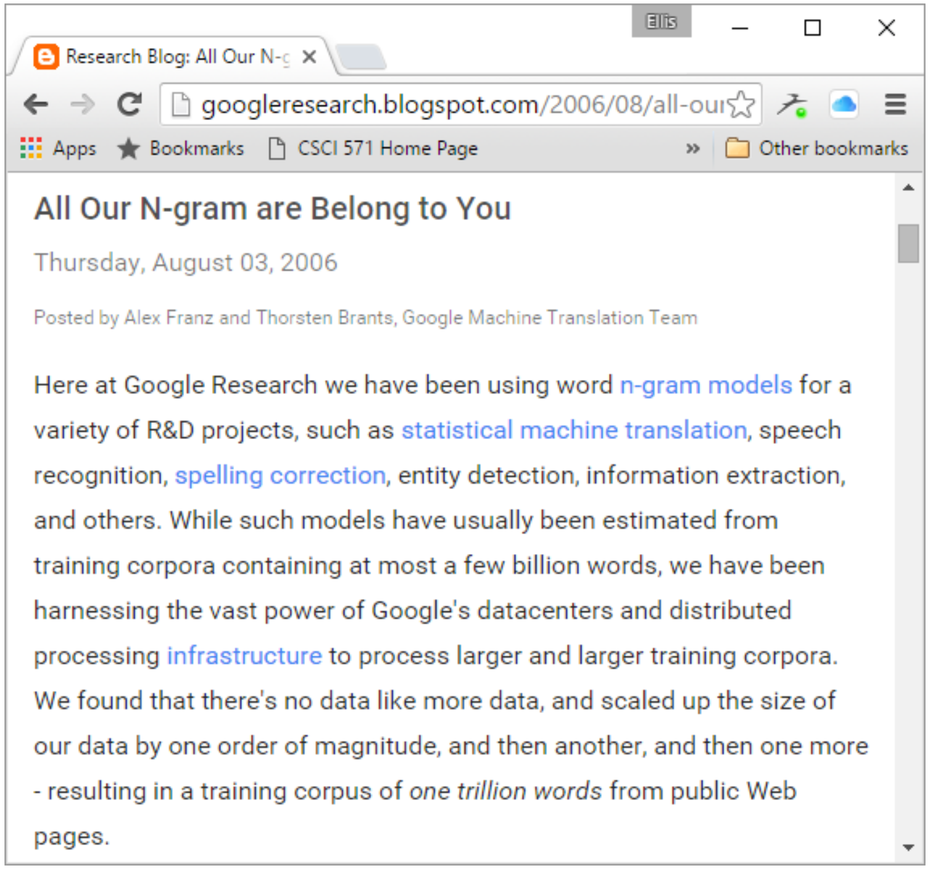
Google Relies Upon *n*-gram Indexes

- Google has investigated the use of n-grams stored in its index, n >= 2;
- N-grams of all lengths form a **Zipf distribution (power law)** with a few common phrases occurring very frequently and a large number occurring with frequency 1
- Google has released the set of n-grams it has determined

Google's N-Gram Database Facts

- Google made available a file of n-grams derived from the web pages it indexed
- http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html
- Statistics for the Google n-gram sample

• Number of tokens	1,024,908,267,229
(1 trillion, 24 billion, 908 million, . . .)	
• Number of sentences	95,119,665,584
• Number of unigrams	13,588,391
• Number of bigrams	314,843,401
• Number of trigrams	977,069,902
• Number of four grams	1,313,818,354
• Number of five grams	1,176,470,663


Research Blog: All Our N-g

googleresearch.blogspot.com/2006/08/all-our

Apps Bookmarks CSCI 571 Home Page Other bookmarks

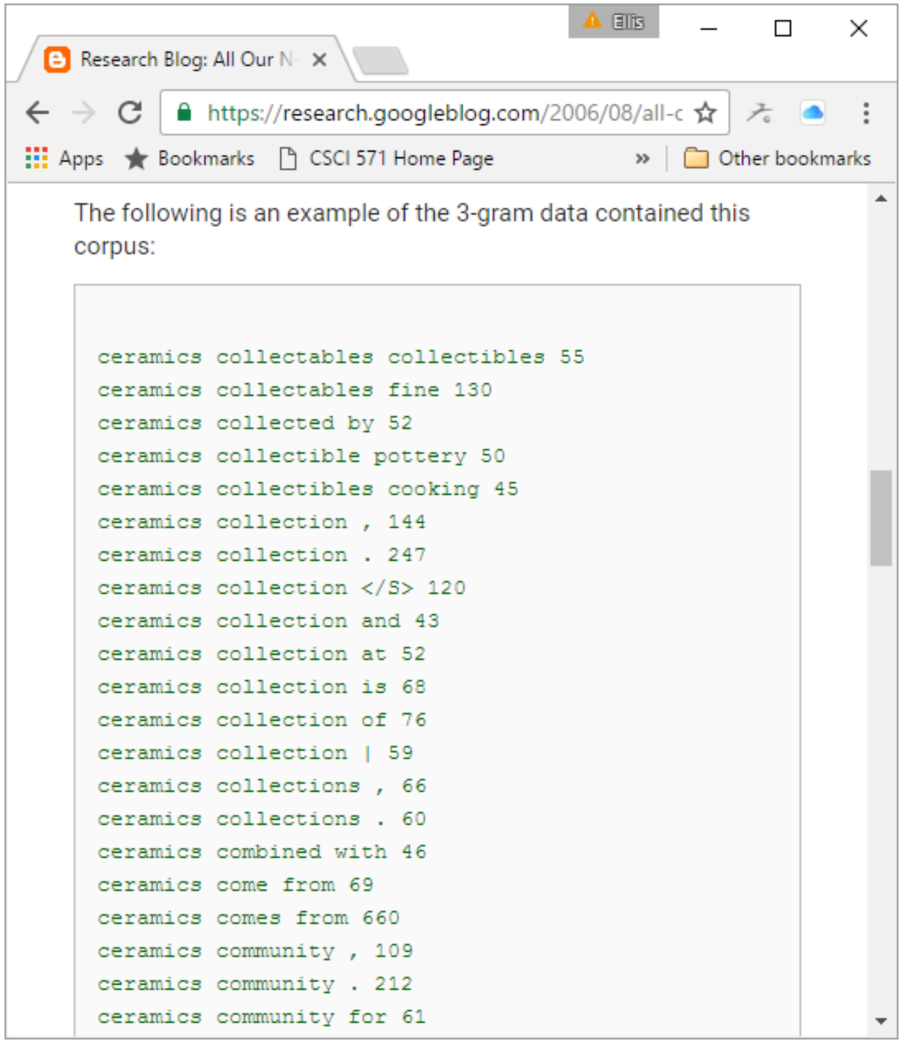
All Our N-gram are Belong to You

Thursday, August 03, 2006

Posted by Alex Franz and Thorsten Brants, Google Machine Translation Team

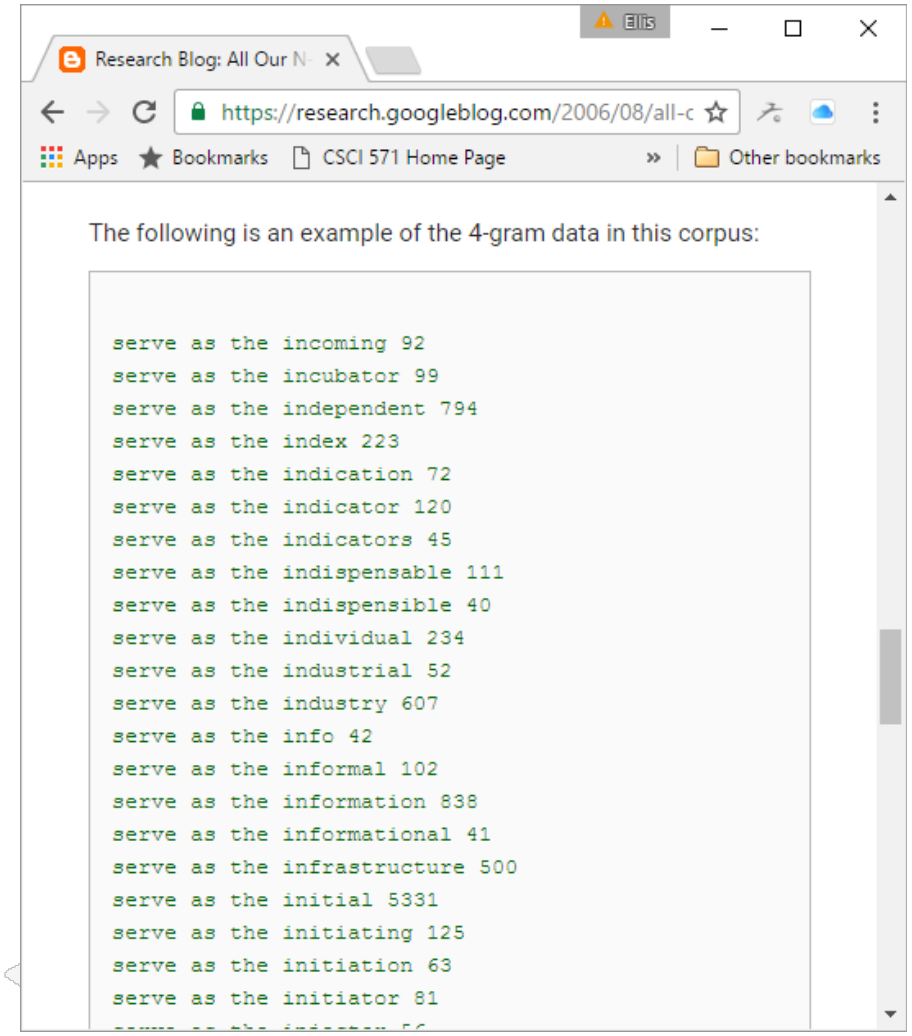
Here at Google Research we have been using word n-gram models for a variety of R&D projects, such as statistical machine translation, speech recognition, spelling correction, entity detection, information extraction, and others. While such models have usually been estimated from training corpora containing at most a few billion words, we have been harnessing the vast power of Google's datacenters and distributed processing infrastructure to process larger and larger training corpora. We found that there's no data like more data, and scaled up the size of our data by one order of magnitude, and then another, and then one more - resulting in a training corpus of *one trillion words* from public Web pages.

Examples of 3-Gram, 4-Gram Data



The following is an example of the 3-gram data contained this corpus:

```
ceramics collectables collectibles 55
ceramics collectables fine 130
ceramics collected by 52
ceramics collectible pottery 50
ceramics collectibles cooking 45
ceramics collection , 144
ceramics collection . 247
ceramics collection </S> 120
ceramics collection and 43
ceramics collection at 52
ceramics collection is 68
ceramics collection of 76
ceramics collection | 59
ceramics collections , 66
ceramics collections . 60
ceramics combined with 46
ceramics come from 69
ceramics comes from 660
ceramics community , 109
ceramics community . 212
ceramics community for 61
```



The following is an example of the 4-gram data in this corpus:

```
serve as the incoming 92
serve as the incubator 99
serve as the independent 794
serve as the index 223
serve as the indication 72
serve as the indicator 120
serve as the indicators 45
serve as the indispensable 111
serve as the indispensible 40
serve as the individual 234
serve as the industrial 52
serve as the industry 607
serve as the info 42
serve as the informal 102
serve as the information 838
serve as the informational 41
serve as the infrastructure 500
serve as the initial 5331
serve as the initiating 125
serve as the initiation 63
serve as the initiator 81
```

The Value of N-Grams

- if we assign a probability to the occurrence of an N-gram or the probability of a word occurring next in a sequence of words, it can be very useful
  - it can help in deciding which N-grams can be chunked together to form single entities (like “San Francisco” chunked together as one word).
  - It can also help make next word predictions. Say you have the partial sentence “Please hand over your”. Then it is more likely that the next word is going to be “test” or “assignment” or “paper” than the next word being “school”.
  - It can help to make spelling error corrections. For instance, the sentence “drink cofee” could be corrected to “drink coffee” if you knew that the word “coffee” had a high probability of occurrence after the word “drink” and also the overlap of letters between “cofee” and “coffee” is high.

An N-Gram Example

- **Consider the 5 sentences**

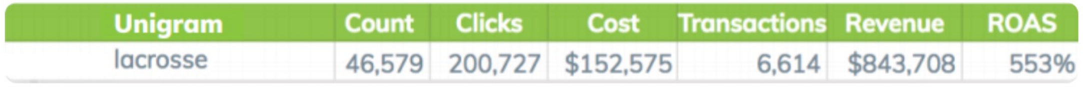
1. He said thank you.
2. He said bye as he walked through the door.
3. He went to San Diego.
4. San Diego has nice weather.
5. It is raining in San Francisco.

- **Assume a bi-gram model, i.e. we will find the probability of a word based only on its previous word**
- **Probability (word) = (the number of times the previous word, pw, occurs before the actual word, aw) / (total number of times pw occurs in the corpus)**
- **(count (pw aw)) / (count (pw))**
- **The probability of “you” following “thank”, or P (you | thank) =**
- **(No. of times “thank you” occurs) / (no. of time “Thank” occurs)**
- **= 1/1 = 1**

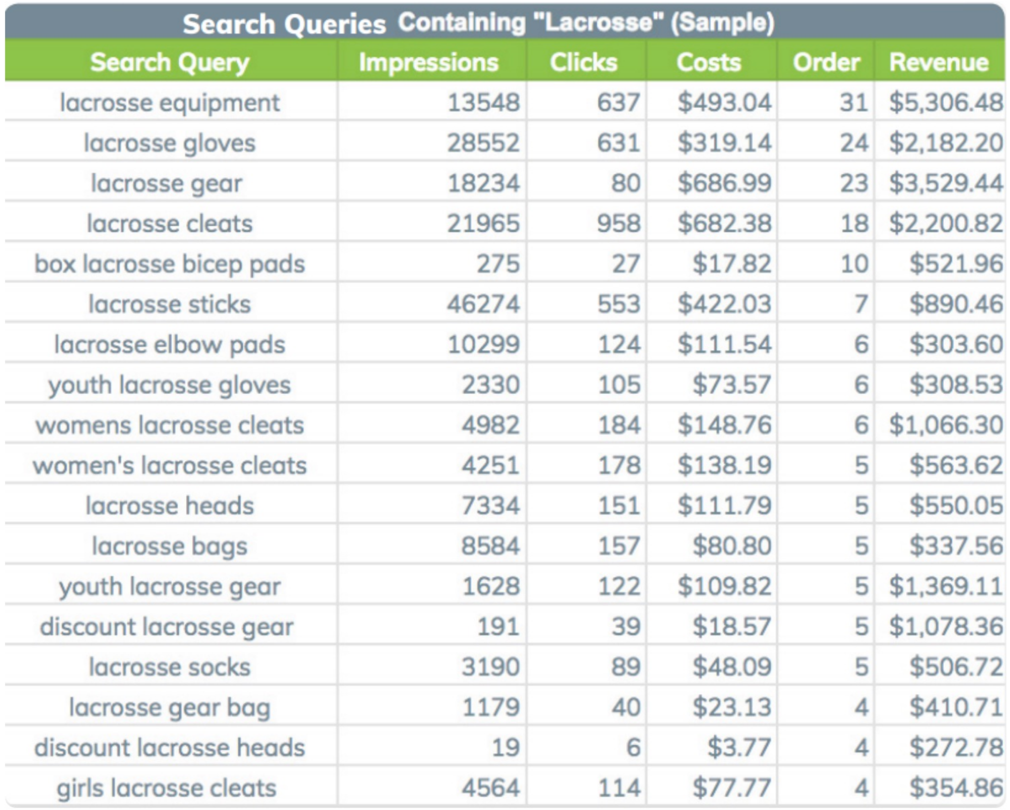
The probability of “Diego” coming after “San” = (No. times “San Diego” occurs) / (No. of times “San” occurs)
= 2/3 = 0.67

N-Grams Used To Find Best and Worst Search Queries

- Lacrosse occurs in 46,579 search terms on Google;
- Overall it produced $843,708 revenue by spending $152,575
- "Lacrosse equipment" was the most profitable, while "girls lacrosse cleats the least



Unigram	Count	Clicks	Cost	Transactions	Revenue	ROAS
lacrosse	46,579	200,727	$152,575	6,614	$843,708	553%

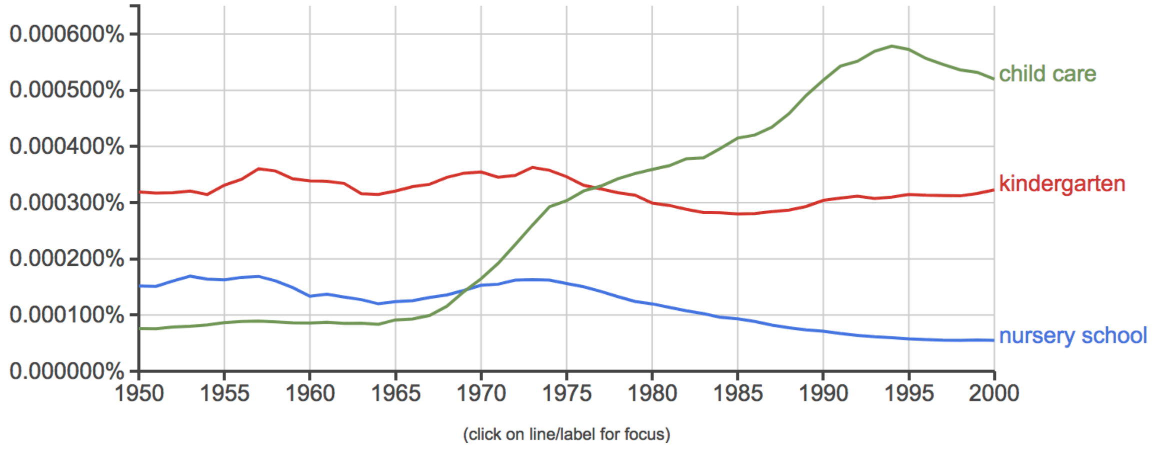


Search Queries Containing "Lacrosse" (Sample)

Search Query	Impressions	Clicks	Costs	Order	Revenue
lacrosse equipment	13548	637	$493.04	31	$5,306.48
lacrosse gloves	28552	631	$319.14	24	$2,182.20
lacrosse gear	18234	80	$686.99	23	$3,529.44
lacrosse cleats	21965	958	$682.38	18	$2,200.82
box lacrosse bicep pads	275	27	$17.82	10	$521.96
lacrosse sticks	46274	553	$422.03	7	$890.46
lacrosse elbow pads	10299	124	$111.54	6	$303.60
youth lacrosse gloves	2330	105	$73.57	6	$308.53
womens lacrosse cleats	4982	184	$148.76	6	$1,066.30
women's lacrosse cleats	4251	178	$138.19	5	$563.62
lacrosse heads	7334	151	$111.79	5	$550.05
lacrosse bags	8584	157	$80.80	5	$337.56
youth lacrosse gear	1628	122	$109.82	5	$1,369.11
discount lacrosse gear	191	39	$18.57	5	$1,078.36
lacrosse socks	3190	89	$48.09	5	$506.72
lacrosse gear bag	1179	40	$23.13	4	$410.71
discount lacrosse heads	19	6	$3.77	4	$272.78
girls lacrosse cleats	4564	114	$77.77	4	$354.86

Google's N-Gram Viewer

- The **Google N-Gram Viewer** or **Google Books N-Gram Viewer** is an online search engine that charts frequencies of any set of comma-delimited search strings using a yearly count of n-grams found in sources printed between 1500 and 2008 in **Google's** text corpora in English, Chinese (simplified), French, German, Hebrew, Italian, ...
  - https://books.google.com/ngrams/, or for examples see books.google.com/ngrams/info
- The program can search for a single word or a phrase, including misspellings
- The n-grams are matched with the text within the selected corpus, and, if found in 40 or more books, are then plotted on a graph
- The data used for the search are composed of total_counts, 1-grams, 2-grams, 3-grams, 4-grams, and 5-grams files for each language



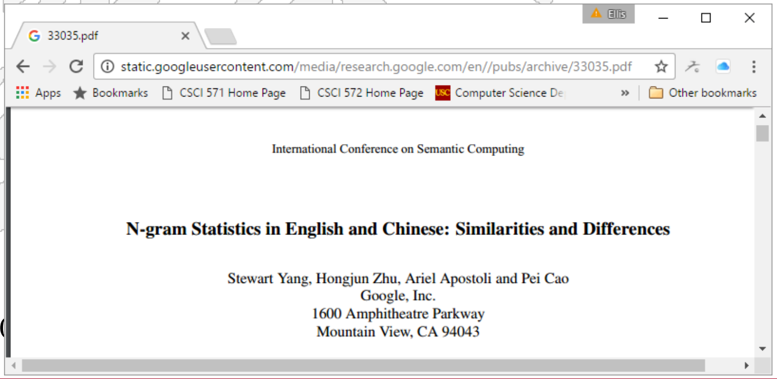
This shows trends in three n-grams from 1950 to 2000:
"nursery school" (a *2-gram* or *bigram*),
"kindergarten" (a *1-gram* or *unigram*),
and
"child care" (another bigram)

USC Viterbi School of Engineering

Comparing N-Grams Across Languages

- S. Yang et al, N-gram statistics in English and Chinese: Similarities and differences, ICSC, 2007, Int'l Conf. on semantic computing, 454-460
- http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/pubs/archive/33035.pdf
- The authors above analyzed 200 million randomly sampled English and Chinese Web pages and concluded:

1. The distribution of the unique number of n-grams is similar between English and Chinese, though the Chinese distribution is shifted to larger N
2. The distribution indicates that on average 1.5 Chinese characters correspond to 1 English word
3. While frequency distributions of uni-grams and bi-grams are very different, the frequency distribution for 3-grams and 4-grams are strikingly similar



International Conference on Semantic Computing

N-gram Statistics in English and Chinese: Similarities and Differences

Stewart Yang, Hongjun Zhu, Ariel Apostoli and Pei Cao
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043