Inverted indexing

●●

USC Viterbi School of Engineering

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

 2

●●

USC Viterbi School of Engineering

Creating an Inverted Index

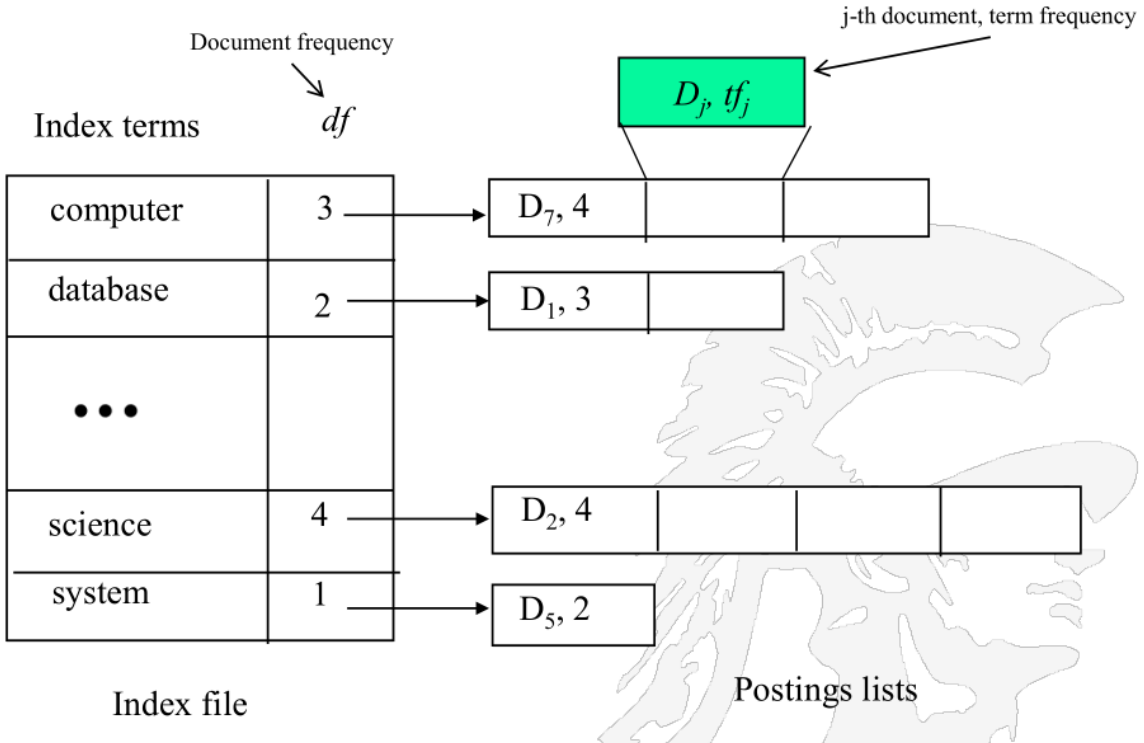
- An inverted index is typically composed of a vector containing all distinct words of the text collection in lexicographical order (which is called the **vocabulary**) and for each word in the vocabulary, a list of all documents (and text positions) in which that word occurs
  - This is nothing more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

 3

●●

USC Viterbi School of Engineering

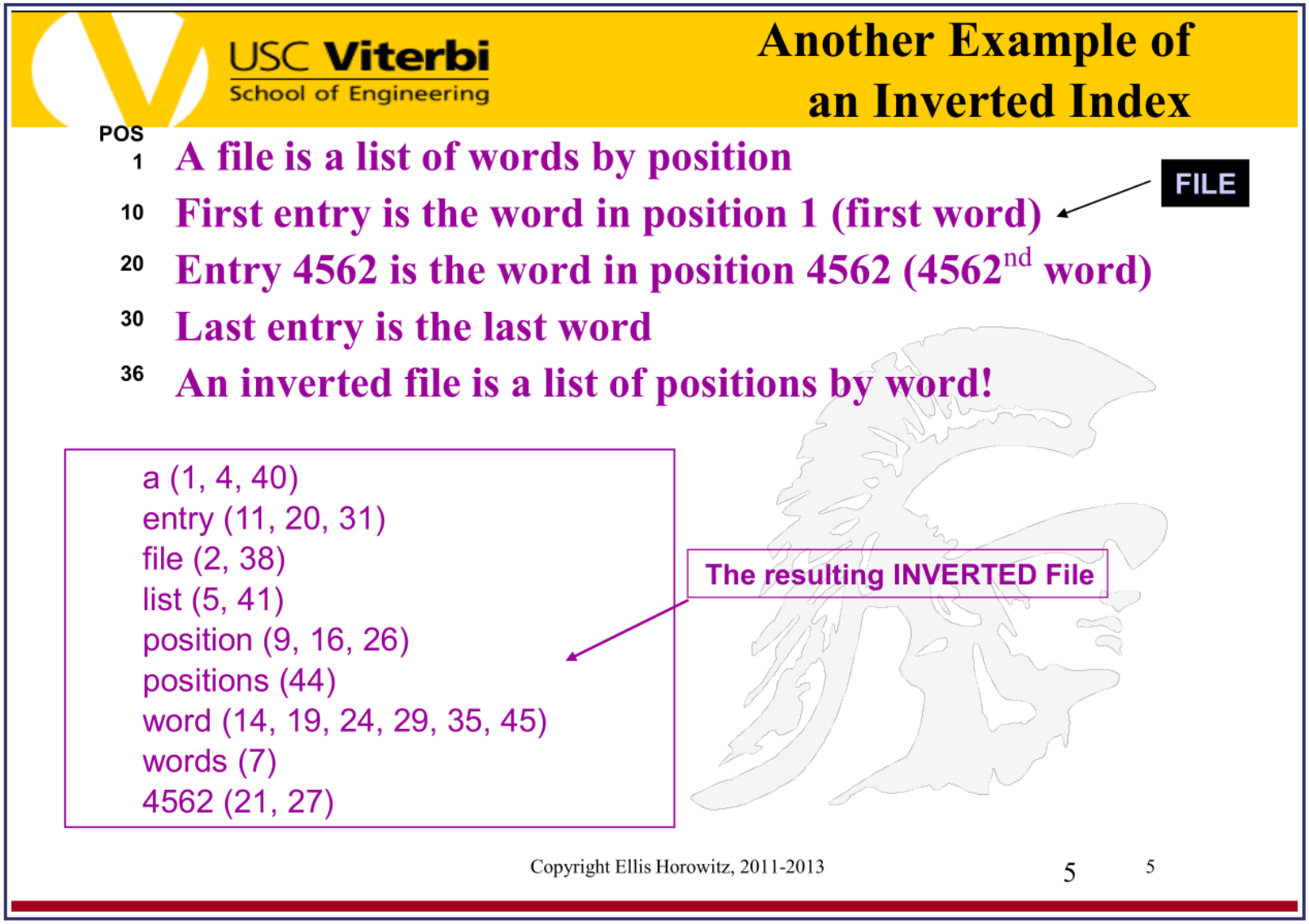
Inverted Index Example



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

4

●●


USC Viterbi
School of Engineering
Another Example of an Inverted Index
POS
1 A file is a list of words by position
10 First entry is the word in position 1 (first word)
20 Entry 4562 is the word in position 4562 (4562nd word)
30 Last entry is the last word
36 An inverted file is a list of positions by word!
FILE
a (1, 4, 40)
entry (11, 20, 31)
file (2, 38)
list (5, 41)
position (9, 16, 26)
positions (44)
word (14, 19, 24, 29, 35, 45)
words (7)
4562 (21, 27)
The resulting INVERTED File
Copyright Ellis Horowitz, 2011-2013
5
5

Yet another example (from the book):

Draw the inverted index that would be built for the following document collection. (See Figure 1.3 for an example.)

Doc 1 new home sales top forecasts
Doc 2 home sales rise in july
Doc 3 increase in home sales in july
Doc 4 july new home sales rise

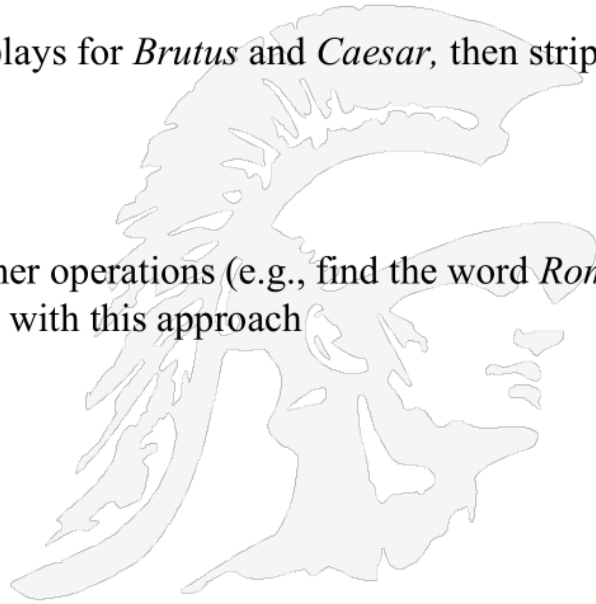
SOLUTION. Inverted Index: forecast->1 home->1->2->3->4 in->2->3 increase->3 july->2->3 new->1->4 rise->2->4 sale->1->2->3->4 top->1

●●

USC Viterbi School of Engineering

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



●●

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

Brutus *AND* ***Caesar*** but *NOT* ***Calpurnia***

1 if play contains word, 0 otherwise

••

USC Viterbi School of Engineering

Incidence Vectors

- So we have a 0/1 vector for each term.
- To answer 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

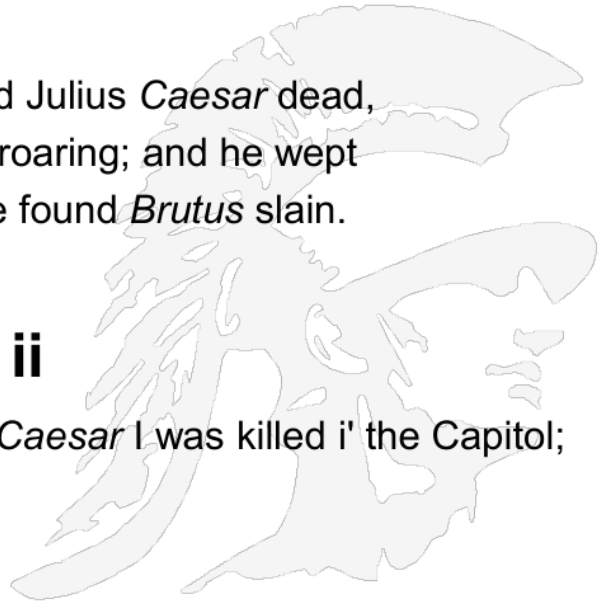
●●

USC Viterbi School of Engineering

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.

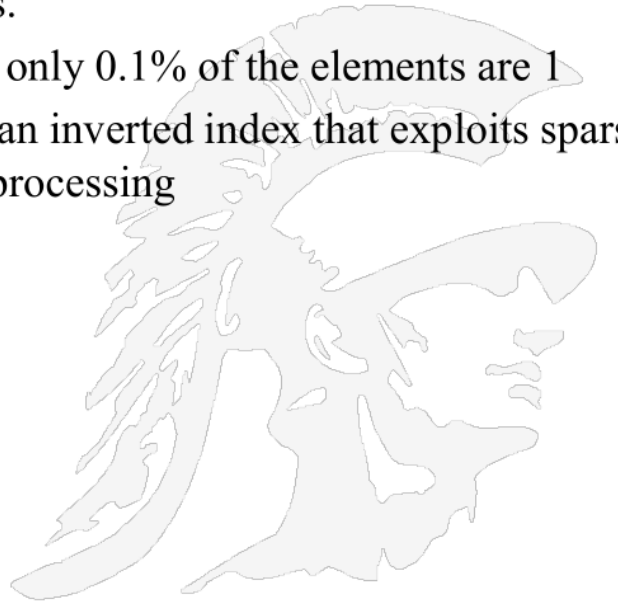


●●

USC Viterbi School of Engineering

Inverted Indexes 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 has 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

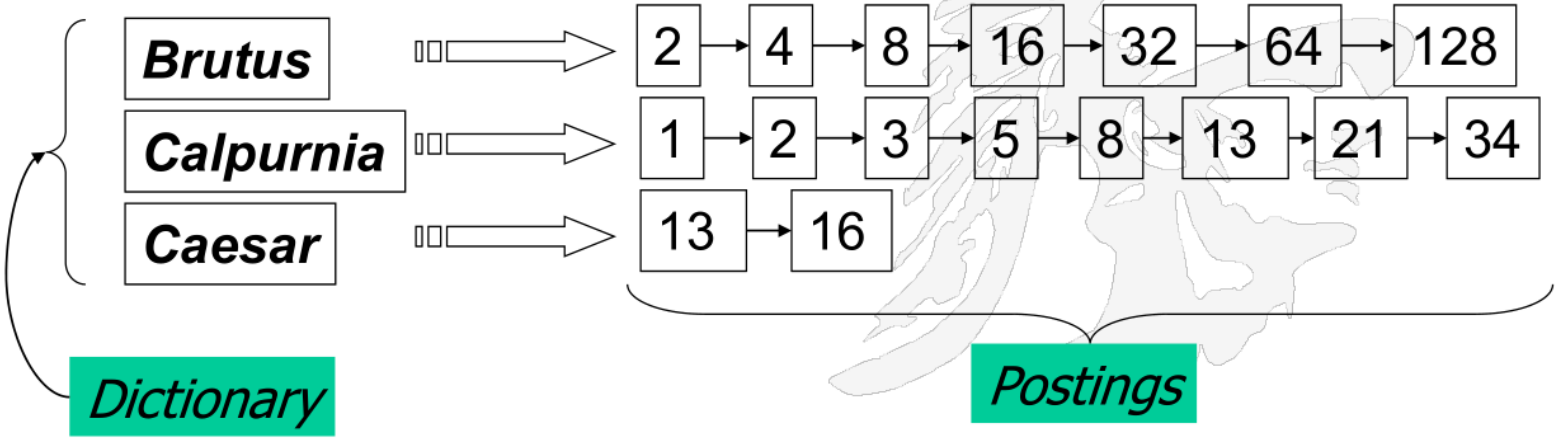


●●

USC Viterbi School of Engineering

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
  - Dynamic space allocation
  - Insertion of terms into documents easy
  - However, there is space overhead of pointers, though this is not too serious



 11

••

USC Viterbi School of Engineering

Inverted Index

- Documents are parsed to extract words and these are saved with the document ID i.e a sequence of (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
	12

●●

USC Viterbi School of Engineering

- 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 WWW, documents are constantly being added and the terms are constantly increasing

13

●●

USC Viterbi School of Engineering

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

Why frequency? Will discuss later.

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

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

 14

●●

USC Viterbi School of Engineering

- **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

Copyright Ellis Horowitz, 2011-2013 15

••

USC Viterbi School of Engineering

Query Processing Across the Postings List

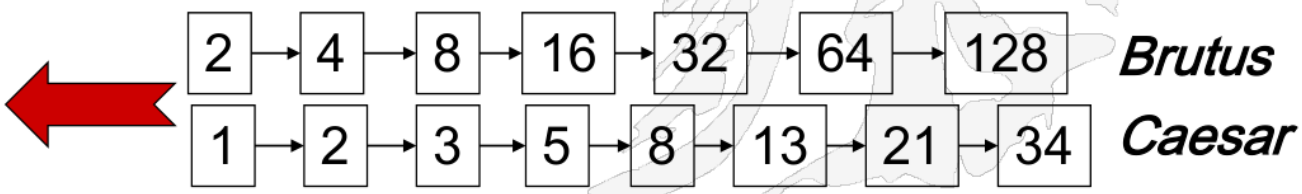
- **Consider processing the query:**

  Brutus AND Caesar

  – Locate *Brutus* in the Dictionary;
    - Retrieve its postings.

  – Locate *Caesar* in the Dictionary;
    - Retrieve its postings.

  – "Merge" the two postings (postings are document ids):

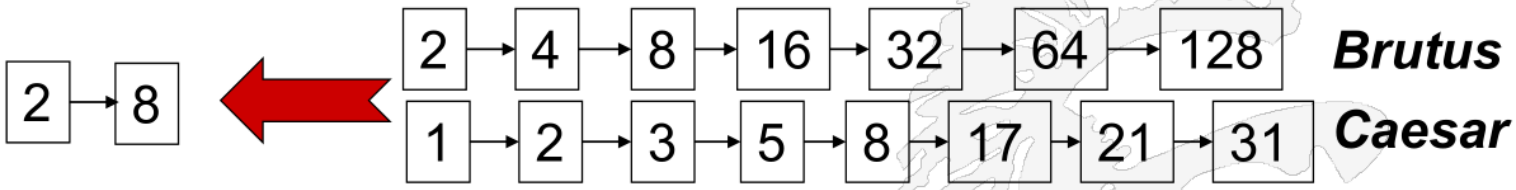


••

USC Viterbi School of Engineering

Basic Merge

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



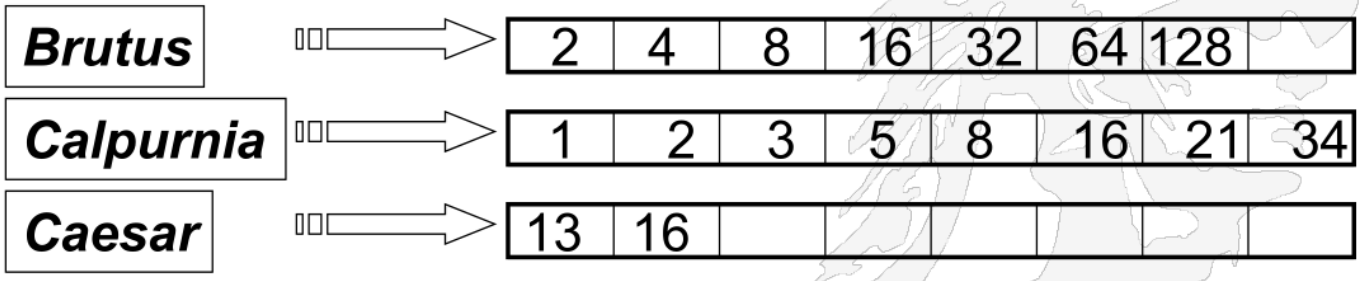
If the list lengths are *m* and *n*, the merge takes O(*m*+*n*) operations.

••

USC Viterbi School of Engineering

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***

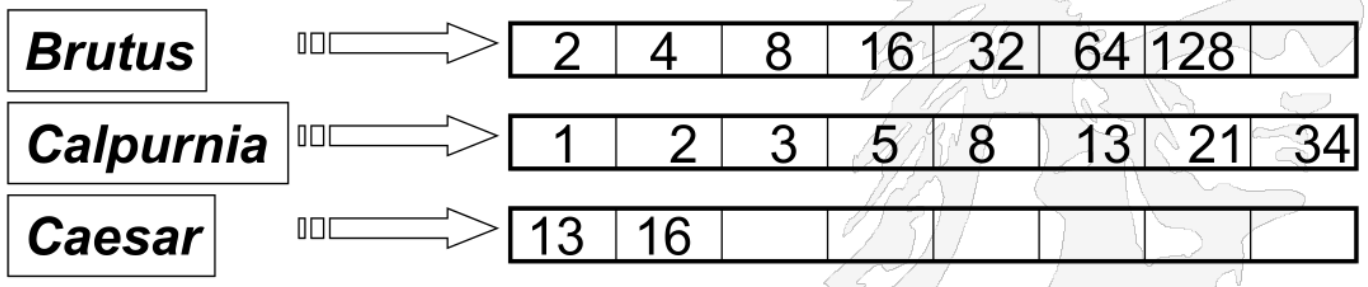
••

USC Viterbi School of Engineering

Query Optimization Example

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

This is why we kept freq in dictionary



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

 19

●●

USC Viterbi
School of Engineering

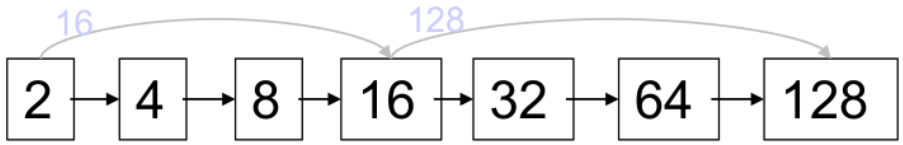
To speed up the merging of postings we use the technique of ***Skip Pointers***

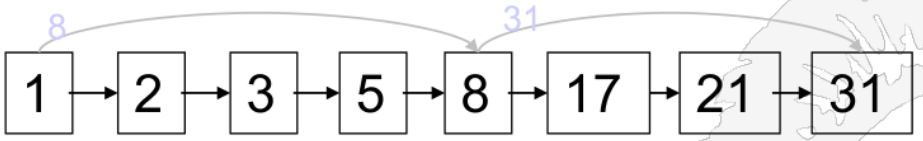
••

USC Viterbi School of Engineering

The Technique of Skip Pointers

Augment postings with skip pointers (at indexing time)





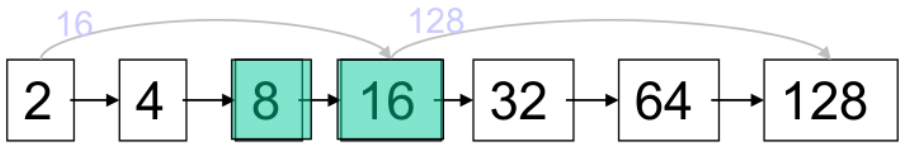
- **Why?**
- **To skip postings that will not figure in the search results.**
- **How?**
- **Where do we place skip pointers?**

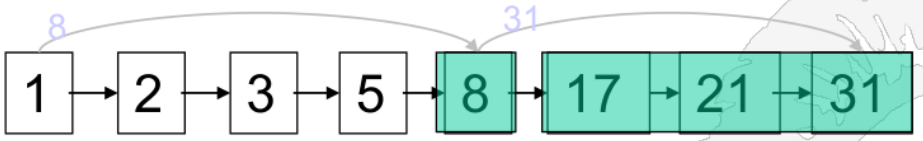
21

••

USC Viterbi School of Engineering

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.

22

Aside: skip pointers (aka skip lists) can be used to search a linked list of sorted items faster than O(n), ie. in O(sqrt(n))...

●●

USC Viterbi School of Engineering

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?

- 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.

- 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.

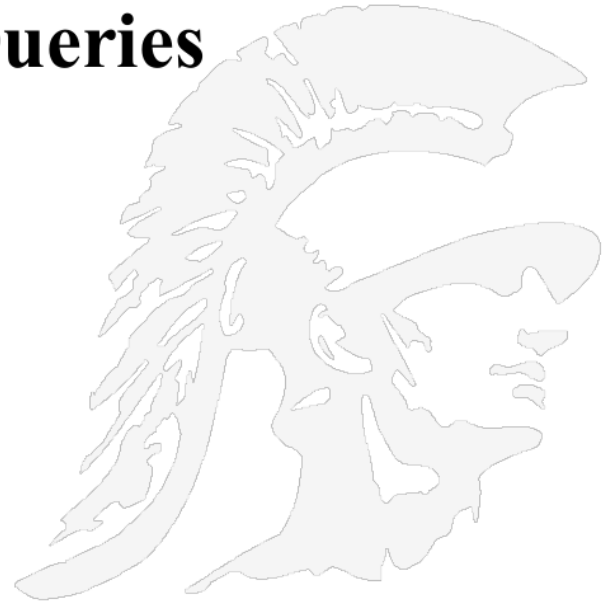
- Building effective skip pointers is easy if an index is relatively static; it is harder if a postings list keeps changing because of updates. A malicious deletion strategy can render skip lists ineffective.

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

●●

USC Viterbi
School of Engineering

Phrase Queries



••

USC Viterbi School of Engineering

Phrase queries

- We want to answer queries such as "*stanford university*" – as a phrase
- Thus the sentence *"I went to university at Stanford"* is not a match.
  - The concept of phrase queries has proven easily understood by users; about 10% of web queries are phrase queries
- No longer suffices to store only

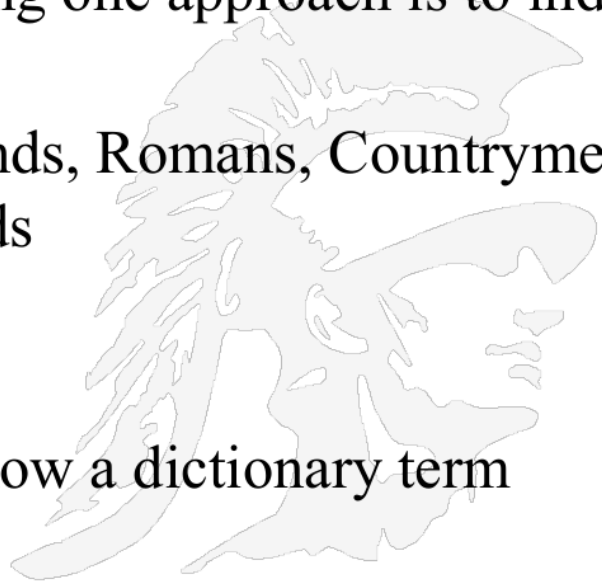
<*term* : *docs*> entries

••

USC Viterbi School of Engineering

Using Biword Indexes for Phrase Searching

- A biword (or a 2-gram) is a consecutive pair of terms in some text
- To improve phrase searching one approach is to index every biword 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 a dictionary term



••

USC Viterbi School of Engineering

Handling Longer Phrase Queries

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

stanford university palo alto

The query can be broken into the Boolean query on biwords:

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 biwords, but not the full 4 word query)

••

USC Viterbi School of Engineering

Alternate Solution Using Positional Indexes

- **Store, for each *term*, entries of the form:**

  <number of docs containing *term*;

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

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

  etc.>

••

USC Viterbi School of Engineering

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, …>

- **Nevertheless, this expands postings storage *substantially***

••

USC Viterbi School of Engineering

Processing a Phrase Query

- Extract inverted index entries for each distinct term: *to, be, or, not.*
- Merge their *doc:position* lists to enumerate all positions with "*to be or not to be*".
  - ***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

●●

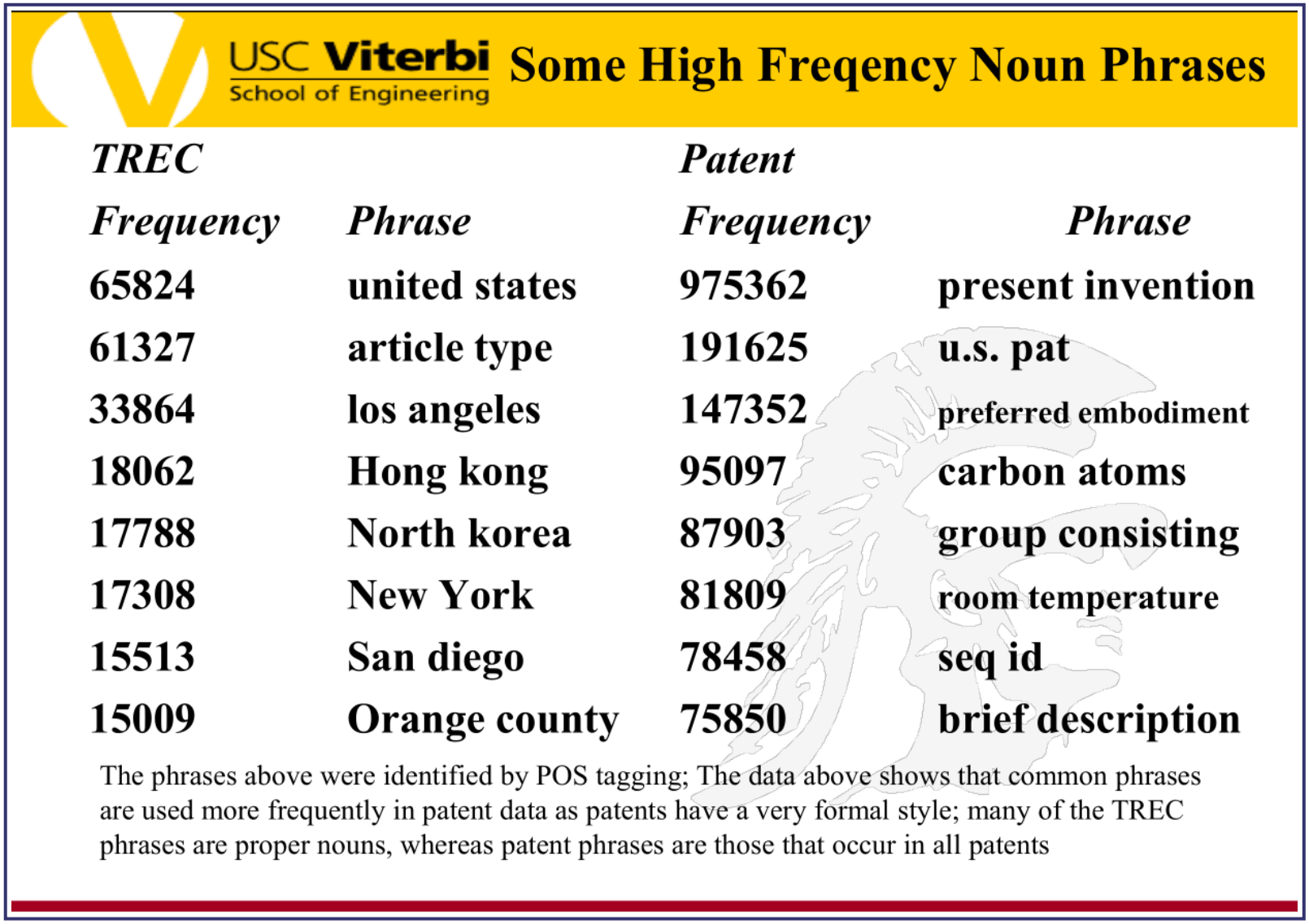
USC Viterbi School of Engineering

Another Approach

- Among possible queries, **nouns and noun phrases** often appear, e.g.

  "abolition of slavery", "renegotiation of the constitution"
- But as seen above, related nouns can often be divided from each other by various function words
- These needs can be incorporated into the biword indexing model in the following way:
  - First, we tokenize the text and perform **part-of-speech-tagging**. We can then group terms into nouns, including proper nouns, (N) and function words, including articles and prepositions, (X), among other classes.
  - Now deem any string of terms of the form NX*N to be an extended biword. Each such extended biword is made a term in the vocabulary.
  - E.g. "renegotiation of the constitution" is mapped to N X X N (two nouns and two others), so the others are ignored and the two word phrase "renegotiation constitution" is added to the index
- Programs that identify a word's part-of-speech tag are based on statistical or rule-based approaches and are trained using large corpora

••


USC Viterbi
School of Engineering
Some High Freqency Noun Phrases
TREC
Frequency
Phrase
65824
united states
61327
article type
33864
los angeles
18062
Hong kong
17788
North korea
17308
New York
15513
San diego
15009
Orange county
Patent
Frequency
Phrase
975362
present invention
191625
u.s. pat
147352
preferred embodiment
95097
carbon atoms
87903
group consisting
81809
room temperature
78458
seq id
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

••

USC Viterbi School of Engineering

Building *n*-gram Indexes

- Generalizing from bi-words, an ***n-gram*** is any sequence of *n* consecutive words
- N-grams can be identified at the time of parsing
- N-grams of all lengths form a Zipf distribution with a few common phrases occurring very frequently and a large number occurring with frequency 1
- Common n-grams are usually made up of stop words (e.g. “and the” “there is”)
- For each *n*-gram, the inverted index will need pointers to all dictionary terms containing it – the “postings”
- Therefore, the larger the value of *n*, the larger the amount of space required to hold all n-grams

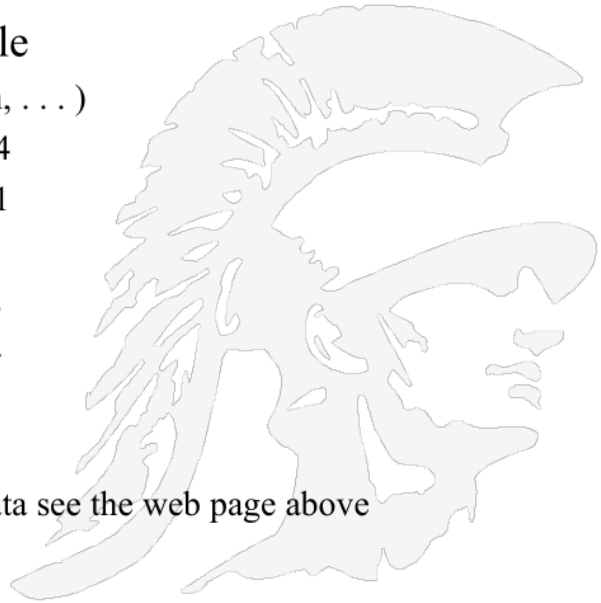
●●

USC Viterbi School of Engineering

Google's N-Gram 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, . . .)
- 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

- For specific examples of 3-gram and 4-gram data see the web page above



●●

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
- They 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

••

USC Viterbi School of Engineering

Distributed Indexing

- For web-scale indexing one
  must use a distributed computing cluster
- Individual machines are fault-prone
  - Can unpredictably slow down or fail
- How do we exploit such a pool of machines?
- Must we maintain a *master* machine directing the indexing job – considered "safe".
- Break up indexing into sets of (parallel) tasks.
- Master machine assigns each task to an idle machine from a pool.

●●

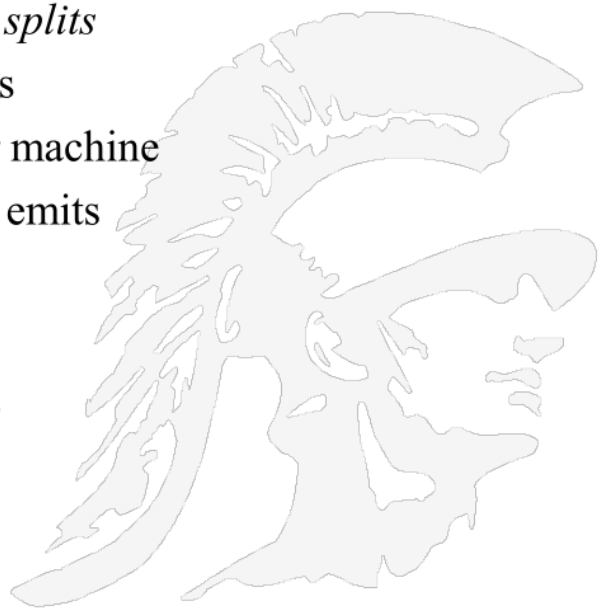
USC Viterbi School of Engineering

Parallel Tasks

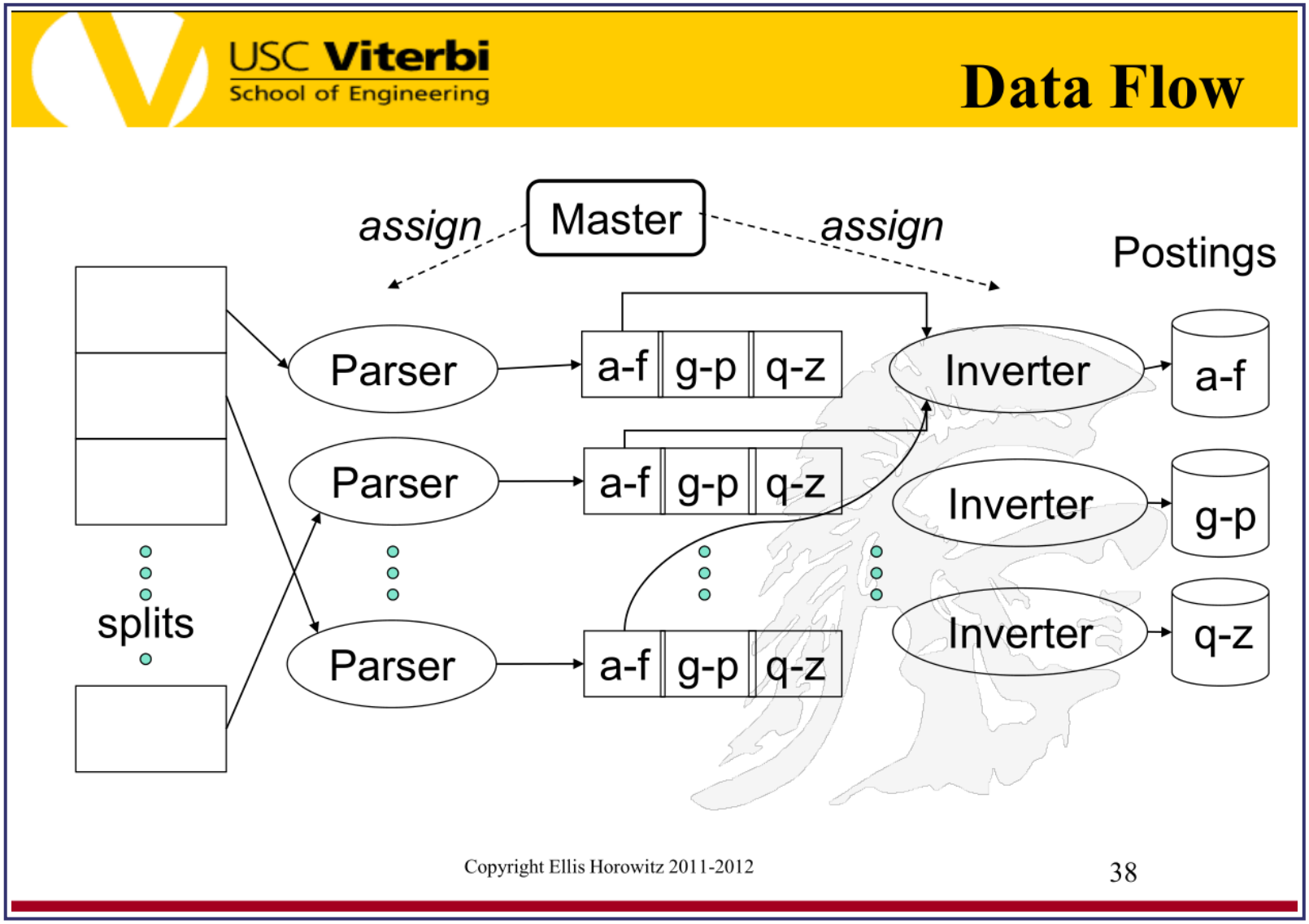
- One approach is to use two sets of parallel tasks
  - Parsers
  - Inverters
- Break the input document corpus into *splits*
  - Each split is a subset of documents
- Master assigns a split to an idle parser machine
- Parser reads a document at a time and emits

(term, doc) pairs

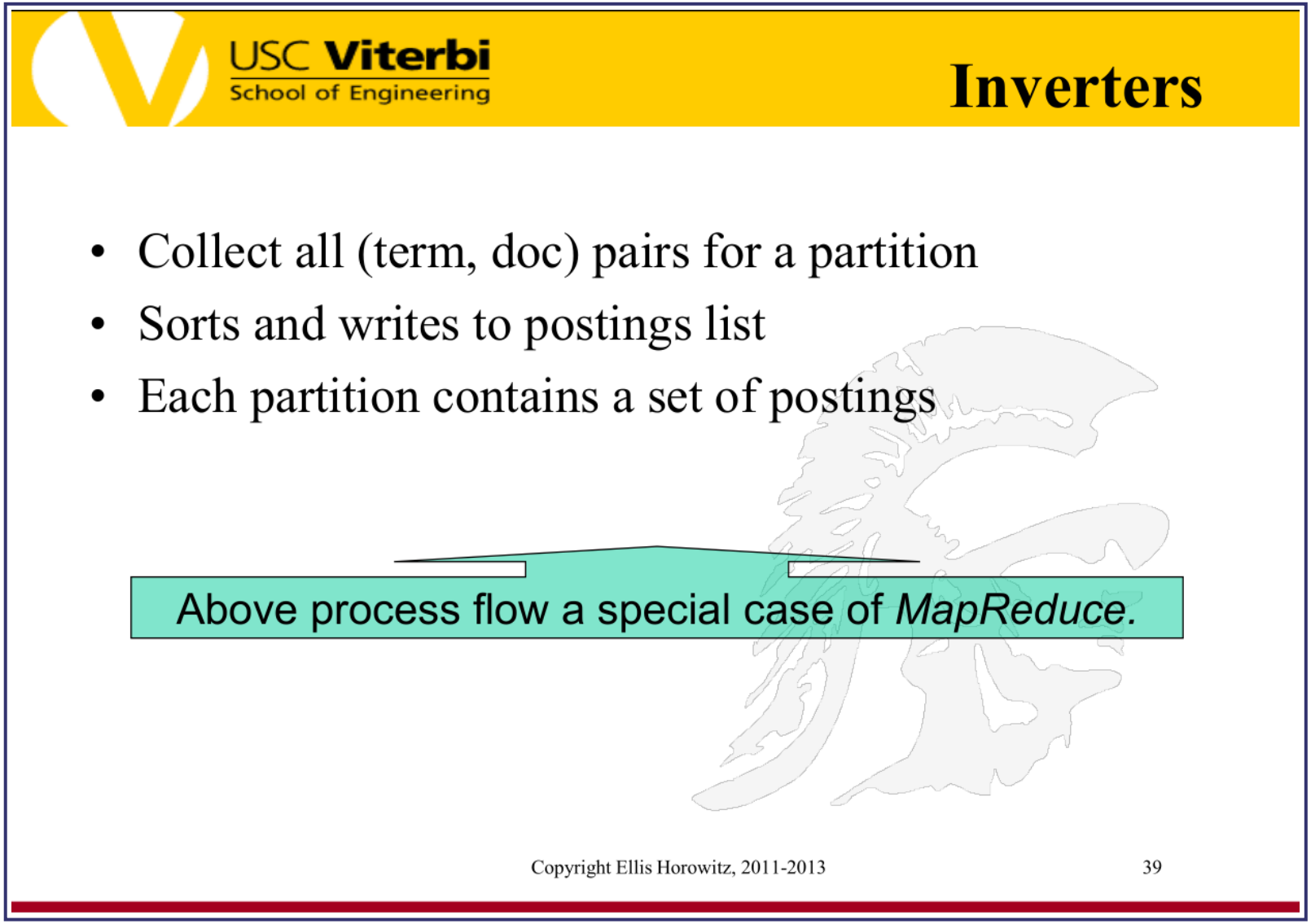
- Parser writes pairs into *j* partitions
- Each for a range of terms' first letters
  - (e.g., *a-f, g-p, q-z*) – here *j*=3.
- Now to complete the index inversion



●●


USC Viterbi
School of Engineering
Data Flow
assign
Master
assign
Postings
Parser
Parser
Parser
a-f g-p q-z
a-f g-p q-z
a-f g-p q-z
Inverter
Inverter
Inverter
a-f
g-p
q-z
splits
Copyright Ellis Horowitz 2011-2012
38

••


USC Viterbi
School of Engineering
Inverters
• Collect all (term, doc) pairs for a partition
• Sorts and writes to postings list
• Each partition contains a set of postings
Above process flow a special case of *MapReduce*.
Copyright Ellis Horowitz, 2011-2013
39

••

USC Viterbi School of Engineering

Simplest Approach to Dynamic Indexing

- Maintain "big" main index
- New docs go into "small" auxiliary index
- Search across both, merge results periodically
- Deletions
  - Invalidation bit-vector for deleted docs
  - Filter docs output on a search result by this invalidation bit-vector
- Periodically, re-index into one main index