# CSCI 5417
# Information Retrieval Systems
Jim Martin

Lecture 4
9/1/2011

## Today

- Finish up spelling correction
- Realistic indexing
  - Block merge
  - Single-pass in memory
  - Distributed indexing
- Next HW details

## Query correction

- Our principal focus here
  - Examples like the query
    - ***Alanas Morisett***
- We can either
  - Retrieve using that spelling
  - Retrieve documents indexed by the correct spelling, OR
  - Return several suggested alternative queries with the correct spelling
    - *Did you mean … ?*
    - This requires an interactive session

## Isolated word correction

- Fundamental premise – there is a lexicon from which the correct spellings come
- Some basic choices for this
  - A standard lexicon such as
    - Webster's English Dictionary
    - An "industry-specific" lexicon – hand-maintained
  - The lexicon derived from the indexed corpus
    - E.g., all words on the web
    - All names, acronyms etc.
      - Including the misspellings?

## Isolated word correction

- Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q
- What does “closest” mean?
- Several alternatives
  - Edit distance
  - Weighted edit distance
  - Bayesian models
  - Character *n*-gram overlap

## Edit distance

- Given two strings $S_1$ and $S_2$, the minimum number of basic operations to covert one to the other
- Basic operations are typically character-level
  - Insert
  - Delete
  - Replace
- E.g., the edit distance from ***cat*** to ***dog*** is 3.
- Generally found by dynamic programming via minimum edit distance

## Weighted edit distance

- As above, but the weight of an operation depends on the character(s) involved
  - Meant to capture keyboard errors, e.g. ***m*** more likely to be mis-typed as ***n*** than as ***q***
  - Therefore, replacing ***m*** by ***n*** is a smaller edit distance than by ***q***
  - (Same ideas usable for OCR, but with different weights)
- Require weight matrix as input
- Modify dynamic programming to handle weights (Viterbi)

## Edit distance to all dictionary terms?

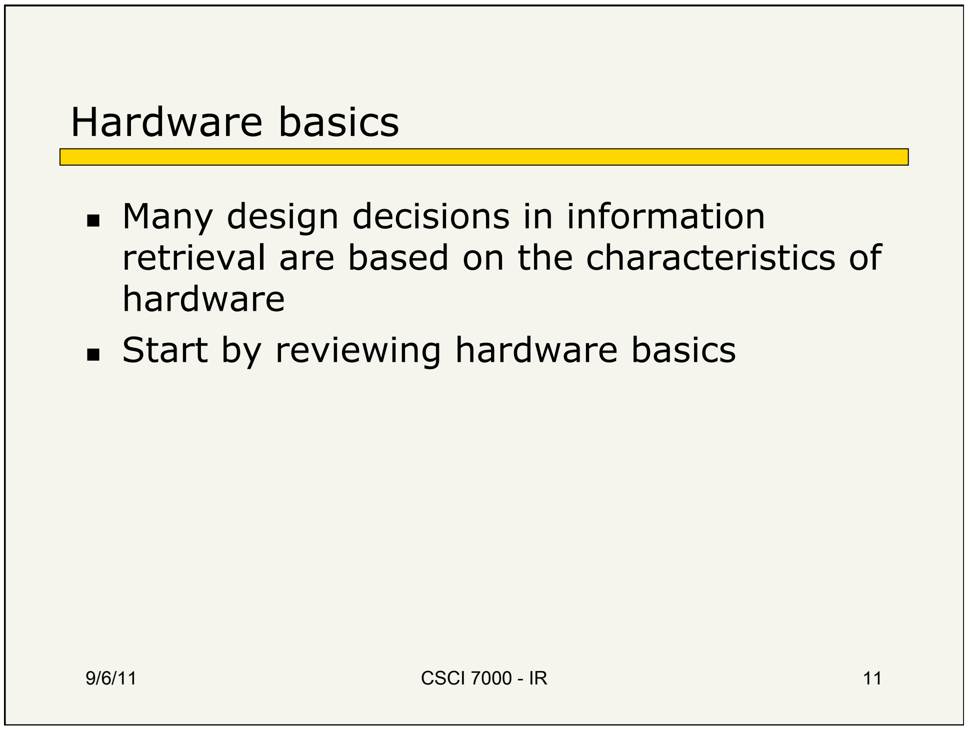
- Given a (misspelled) query – do we compute its edit distance to every dictionary term?
  - Expensive and slow
- How do we cut the set of candidate dictionary terms?
  - Heuristics
    - Assume first letter(s) is correct
    - Character *n*-gram overlap

## General issue in spell correction

- Will enumerate multiple alternatives for "Did you mean"
- Need to figure out which one (or small number) to present to the user
- Use heuristics
  - The alternative hitting most docs
  - Query log analysis + tweaking
    - For especially popular, topical queries
  - Language modeling

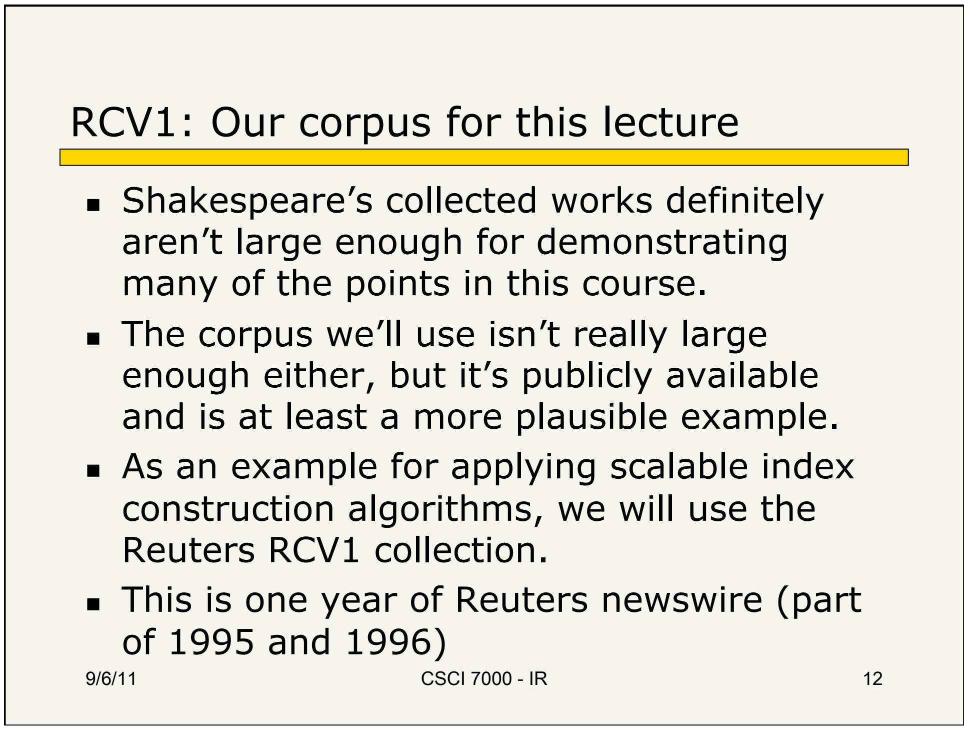
## Back to Index Construction

- How do we construct an index?
- What strategies can we use when there is limited main memory?
  - And there's never enough memory



## Hardware basics

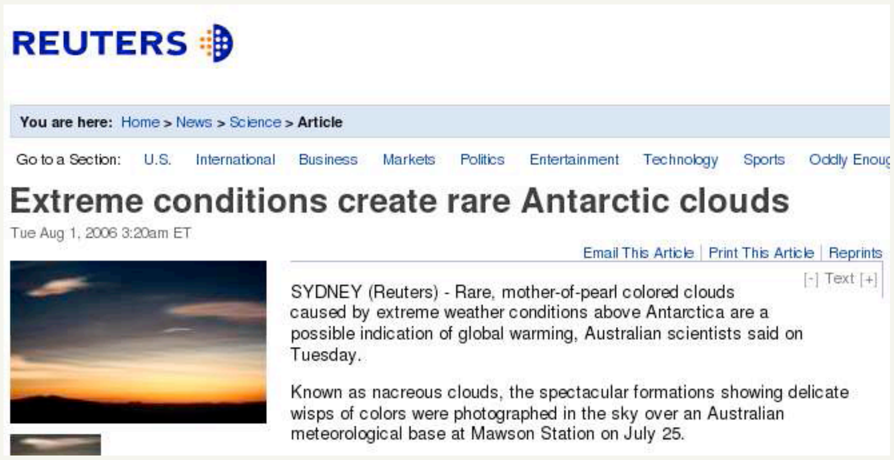
- Many design decisions in information retrieval are based on the characteristics of hardware
- Start by reviewing hardware basics



## RCV1: Our corpus for this lecture

- Shakespeare’s collected works definitely aren’t large enough for demonstrating many of the points in this course.
- The corpus we’ll use isn’t really large enough either, but it’s publicly available and is at least a more plausible example.
- As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.
- This is one year of Reuters newswire (part of 1995 and 1996)

## A Reuters RCV1 document


REUTERS
You are here: Home > News > Science > Article
Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug
Email This Article | Print This Article | Reprints
[-] Text [+]

Extreme conditions create rare Antarctic clouds

Tue Aug 1, 2006 3:20am ET

SYDNEY (Reuters) - Rare, mother-of-pearl colored clouds caused by extreme weather conditions above Antarctica are a possible indication of global warming, Australian scientists said on Tuesday.

Known as nacreous clouds, the spectacular formations showing delicate wisps of colors were photographed in the sky over an Australian meteorological base at Mawson Station on July 25.

## Reuters RCV1 statistics

| **symbol** | **statistic** | **value** |
|---|---|---|
| N | documents | 800,000 |
| L | avg. # tokens per doc | 120 |
| M | terms (= word types) | 400,000 |
| | avg. # bytes per token (incl. spaces/punct.) | 6 |
| | avg. # bytes per token (without spaces/punct.) | 4.5 |
| | avg. # bytes per term | 7.5 |
| T | tokens | 100,000,000 |

## Recall simple index construction

- Documents are parsed to extract words and these are saved with the Document ID.

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 |

## Key step

- After all documents have been parsed, the pairs file is sorted by terms.

We focus on this sort step. We have 100M items to sort.

| 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 |

| 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 |

## Scaling Index Construction

- Such in-memory index construction does not scale.
- How can we construct an index for very large collections?
- Taking into account basic hardware constraints
  - Memory, disk, speed etc.

## Sort-based Index construction

- As we build the index, we parse docs one at a time.
  - The final postings for any term are incomplete until the last doc has been processed.
  - At 12 bytes per postings entry, demands a lot of space for large collections.
    - Term-id, doc-id, freq (4+4+4)
- T = 100,000,000 in the case of RCV1
  - This can be done in memory now, but typical collections are much larger. E.g. *New York Times* provides index of >150 years of newswire
- So we'll need to store intermediate results on disk.

## Use the same algorithm for disk?

- Can we use the same index construction algorithm for larger collections, but by using a disk-based sort instead of memory?
  - No: Sorting T = 100,000,000 records on disk is too slow – too many disk seeks.
- We need an better idea

## BSBI: Blocked Sort-Based Indexing

- 12-byte (4+4+4) records *(termid, docid, freq)*
- These are generated as we parse docs
- Must now sort 100M such 12-byte records by *term*
- Define a Block ~ 10M such records
  - Can easily fit a couple into memory
  - Say we have 10 such blocks to start with
- Basic idea of algorithm:
  - Accumulate postings for each block, sort, write to disk.
  - Then merge the blocks into one long sorted order.

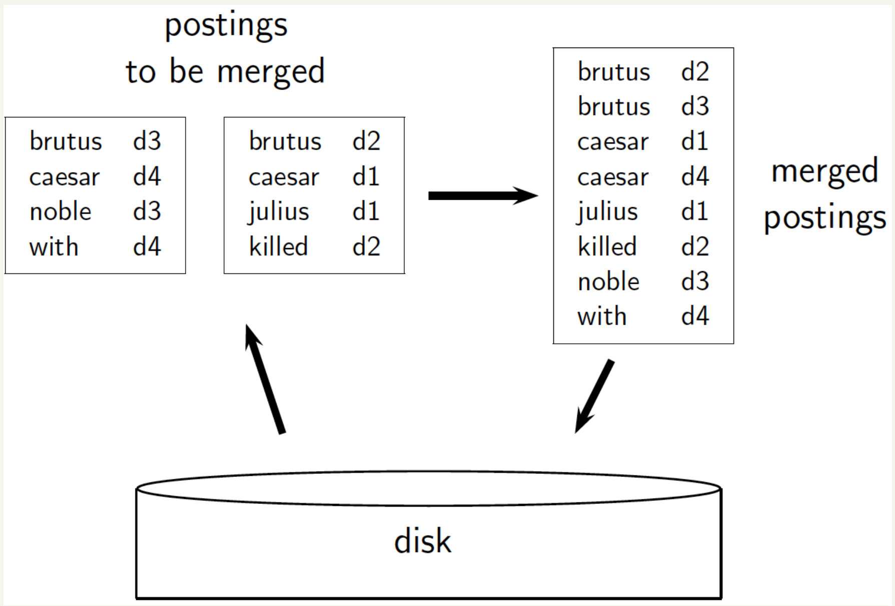
## Dictionary

- This assumes a data-structure to map from terms (strings) to term-ids (ints).
- This dictionary has to be available (in memory) as the blocks are processed to make sure that the terms get assigned the right term-ids
  - That's a structure with 400,000
    - Term strings and Ints
    - Say 20 bytes for the terms, 4 bytes per int

```
BSBIndexConstruction()
1  $n \leftarrow 0$
2  while (all documents have not been processed)
3  do $n \leftarrow n + 1$
4       $block \leftarrow$ ParseNextBlock()
5       BSBI-Invert($block$)
6       WriteBlockToDisk($block, f_n$)
7  MergeBlocks($f_1, \ldots, f_n; f_{\text{merged}}$)
```

## Sorting 10 blocks of 10M records

- First, read each block and sort within:
  - Quicksort takes 2$N$ ln $N$ expected steps
  - In our case 2 x (10M ln 10M) steps
- 10 times this estimate - gives us 10 sorted *runs* of 10M records each.



## Ice Cream

- CS Colloquium Today
    - Thursday 3:30pm in ECCR 265
    - Welcome Back, Ice Cream Event
    - Meet the faculty and staff,
    - Learn some trivia about the faculty,
    - Intro student associations

## HW Questions?/Comments

14480

## Main Problem with Sort-Based Algorithm

- Our assumption was we can keep the dictionary in memory...
- We need the dictionary (which grows dynamically) in order to implement a term to termID mapping
- This isn't a problem for 400k terms. But it is a problem for 13B terms.

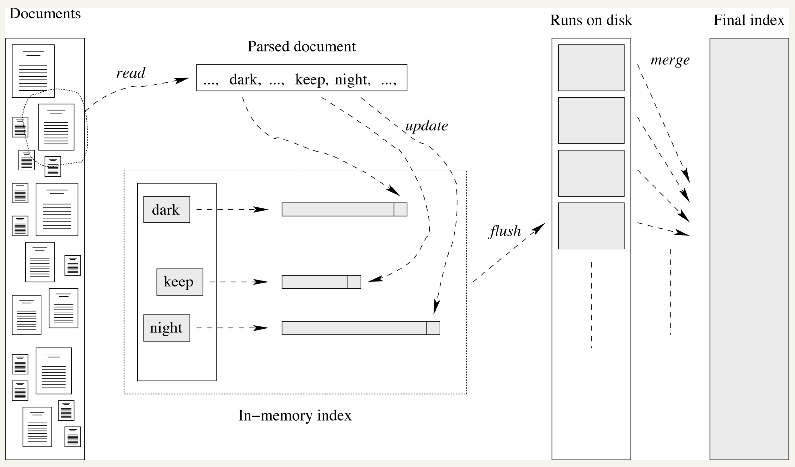
## SPIMI: Single-pass in-memory indexing

- Key idea 1: Generate separate dictionaries for each block – no need to maintain term-termID mapping across blocks; just use terms
- Key idea 2: Don't sort. Accumulate postings in postings lists as they occur.
- With these two ideas we can generate a complete inverted index for each block.
- These separate indexes can then be merged into one big index.

## SPIMI-Invert

```
SPIMI-INVERT(token_stream)
output_file = NEWFILE()
dictionary = NEWHASH()
while (free memory available)
do token ← next(token_stream)
    if term(token) ∉ dictionary
      then postings_list = ADDTODICTIONARY(dictionary, term(token))
      else postings_list = GETPOSTINGSLIST(dictionary, term(token))
    if full(postings_list)
      then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
    ADDTOPOSTINGSLIST(postings_list, docID(token))
sorted_terms ← SORTTERMS(dictionary)
WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
return output_file
```

## Merge algorithm



## Lesson

- The fact that you need a sorted list as output doesn't mean that you need to do a sort... Doing a merge can be good enough.

## Dynamic indexing

- New Docs come in over time
  - postings updates for terms already in dictionary
  - new terms added to dictionary
- Docs can get deleted
- Docs can be altered

## Simplest approach

- Maintain “big” main index
- New docs go into “small” auxiliary index
- Search across both, merge results
- Periodically, re-index into one main index

## Dynamic indexing at search engines

- All the large search engines now do dynamic indexing
- Their indices have frequent incremental changes
  - News items, blogs, new topical web pages
- But (sometimes/typically) they also periodically reconstruct the index from scratch
  - Query processing is then switched to the new index, and the old index is then deleted

## Dynamic Indexing

- That assumes that the “main” index is reasonably static and only needs periodic updates...
- Not true with true real-time indexing
  - Ala twitter

## Distributed indexing

- For web-scale indexing must use a distributed computing cluster
- How do we exploit such a pool of machines?

## Google data centers

- Google data centers mainly contain commodity machines.
- Data centers are distributed around the world (often near cheap power)
- Estimate: a total of 1 million servers, 3 million processors/cores (Gartner 2007)
- Estimate: Google installs 100,000 servers each quarter.
  - Based on expenditures of 200–250 million dollars per year
- About 10% of the computing capacity of the world

## Distributed indexing

- So given a collection distributed across many (thousands of) machines
- Build an index distributed across many (thousands of) machines
- Here we'll look at such indexes distributed across machines by term

## Distributed indexing

- Maintain a *master* machine directing the indexing job – considered “safe”
- Break up indexing into sets of (concurrently executable) tasks
- Master machine assigns each task to an idle machine from a pool

## Parallel tasks

- We will use two sets of parallel tasks
  - *Parsers*
  - *Inverters*
- Break the input document collection 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,docID> pairs

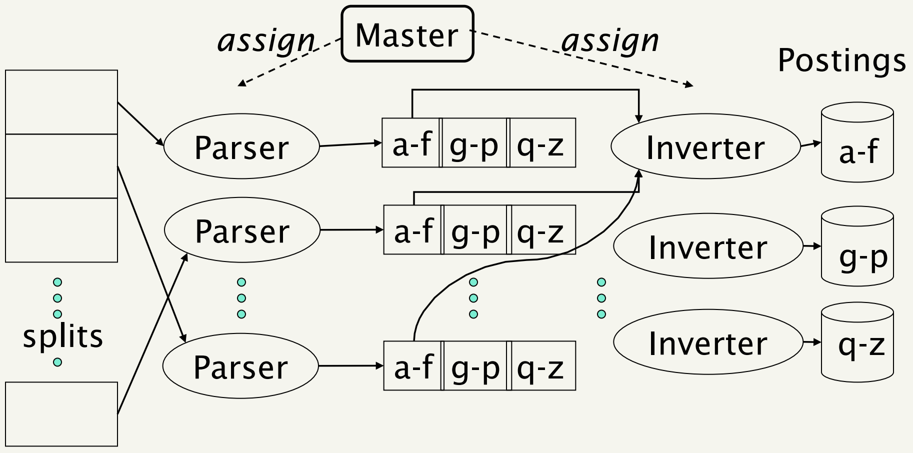
## Parser tasks

- Parser writes pairs into *j* partitions
- One 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

## Inverters

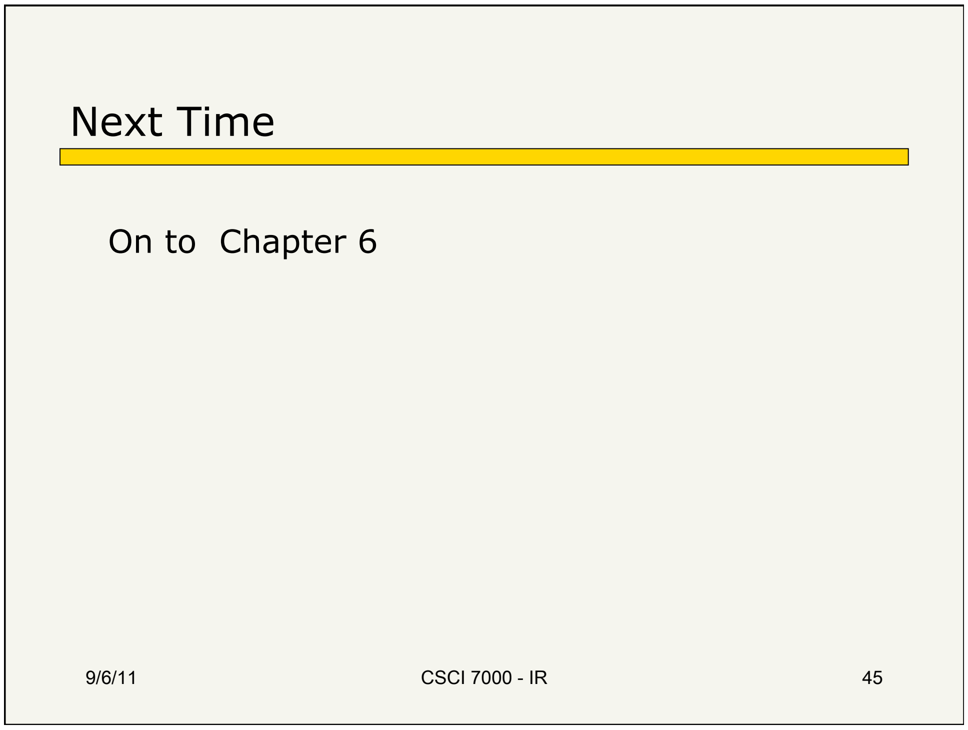
- Collect all (termID, docID) pairs for a partition (from all the parsers)
- Sorts and writes to postings list
- Each partition then contains a set of postings

## Data flow



## MapReduce

- The index construction algorithm we just described is an instance of MapReduce
- MapReduce (Dean and Ghemawat 2004) is a robust and conceptually simple framework for distributed computing …
  - without having to write code for the distribution part.
  - Open source version is called Hadoop



## Next Time

On to  Chapter 6