# Reuters collection example (approximate #'s)

- 800,000 documents from the Reuters news feed
- 200 terms per document
- 400,000 unique terms
- number of postings 100,000,000

REUTERS

You are here: Home > News > Science > Article

Go to a Section: U.S. International Business Markets Politics Entertainment Technology Sports Oddly Enoug

**Extreme conditions create rare Antarctic clouds**

Tue Aug 1, 2006 3:20am ET

Email This Article | Print This Article | Reprints

[-] Text [+]

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 collection example (approximate #'s)

- Sorting 100,000,000 records on disk is too slow because of disk seek time.
  - Parse and build posting entries one at a time
  - Sort posting entries by term
    - Then by document in each term
- Doing this with random disk seeks is too slow
- e.g. If every comparison takes 2 disk seeks and N items need to be sorted with N log2(N) comparisons?
  - 306ish days?

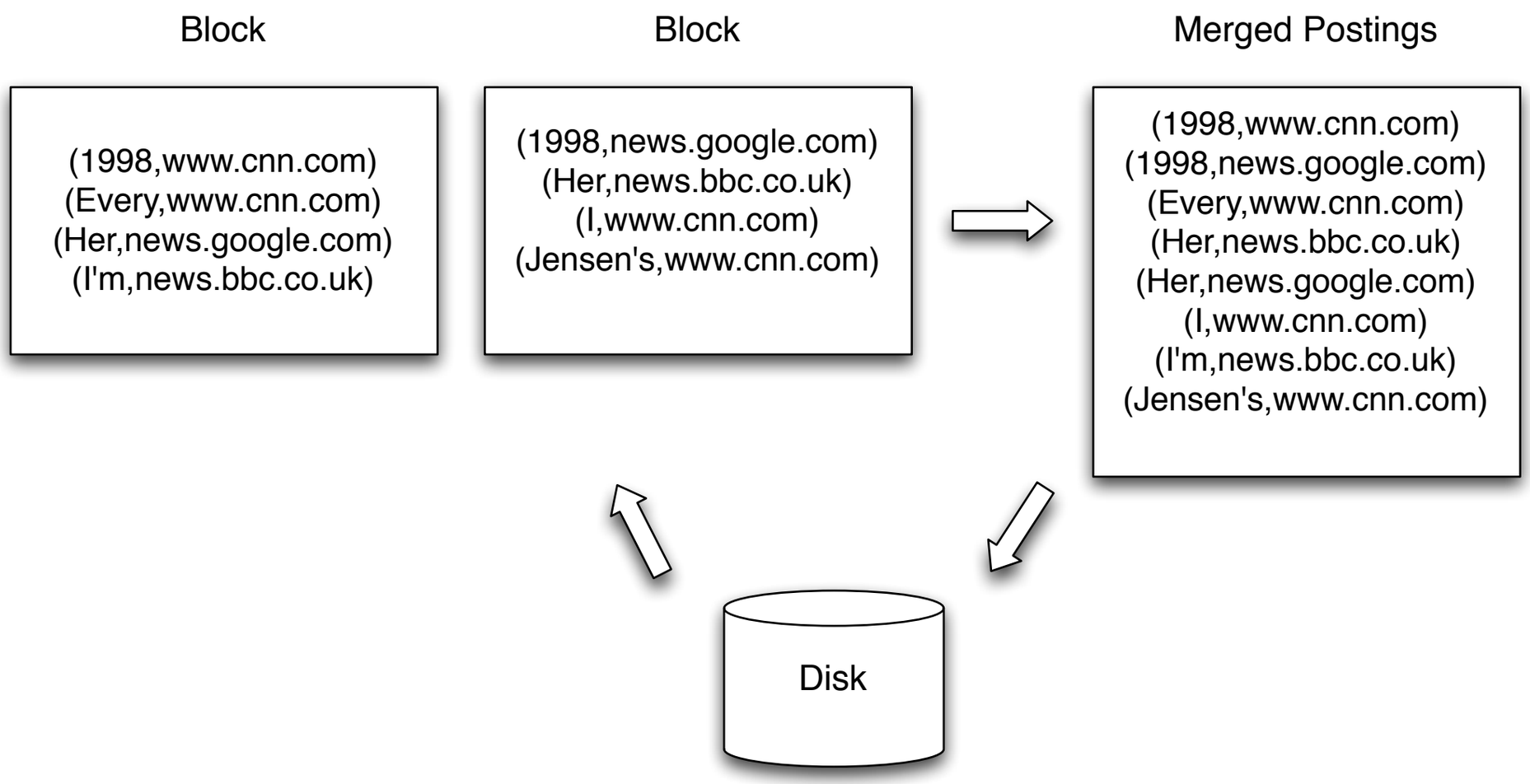
# Reuters collection example (approximate #'s)

- 100,000,000 records
- Nlog2(N) is = 2,657,542,475.91 comparisons
- 2 disk seeks per comparison = 13,287,712.38 seconds x 2
- = 26,575,424.76 seconds
- = 442,923.75 minutes
- = 7,382.06 hours
- = 307.59 days
- = 84% of a year
- = 1% of your life

# Different way to sort index

- 12-byte records  (term, doc, meta-data)
- Need to sort T= 100,000,000 such 12-byte records by term
- Define a block to have 1,600,000 such records
  - can easily fit a couple blocks in memory
  - we will be working with 64 such blocks
- Accumulate postings for each block (real blocks are bigger)
- Sort each block
- Write to disk
- Then merge

# Different way to sort index



## BlockSortBasedIndexConstruction

```
BlockSortBasedIndexConstruction()
 1  n ← 0
 2  while (all documents not processed)
 3      do block ← ParseNextBlock()
 4         BSBI-Invert(block)
 5         WriteBlockToDisk(block, f_n)
 6  MergeBlocks(f_1, f_2..., f_n, f_merged)
```

# Block merge indexing

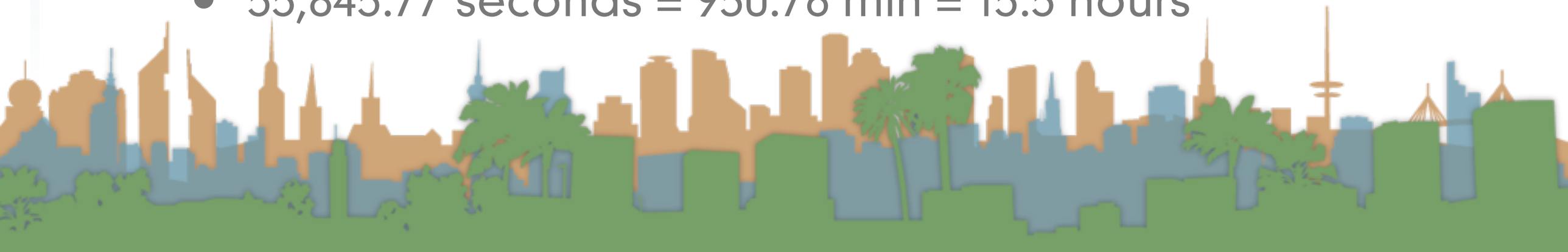
- Parse documents into (TermID, DocID) pairs until "block" is full
- Invert the block
  - Sort the (TermID,DocID) pairs
  - Compile into TermID posting lists
- Write the block to disk
- Then merge all blocks into one large postings file
  - Need 2 copies of the data on disk (input then output)

# Analysis of BSBI

- The dominant term is O(TlogT)
  - T is the number of TermID,DocID pairs
- But in practice ParseNextBlock takes the most time
- Then MergingBlocks
- Again, disk seeks times versus memory access times

# Analysis of BSBI

- 12-byte records  (term, doc, meta-data)
- Need to sort T= 100,000,000 such 12-byte records by term
- Define a block to have 1,600,000 such records
  - can easily fit a couple blocks in memory
  - we will be working with 64 such blocks
- 64 blocks * 1,600,000 records * 12 bytes = 1,228,800,000 bytes
- Nlog2N comparisons is 5,584,577,250.93
- 2 touches per comparison at memory speeds (10e-6 sec) =
  - 55,845.77 seconds = 930.76 min = 15.5 hours



# Overview

- Introduction
- Hardware
- BSBI - Block sort-based indexing
- SPIMI - Single Pass in-memory indexing
- Distributed indexing
- Dynamic indexing
- Miscellaneous topics

# Single-Pass In-Memory Indexing

## SPIMI

- BSBI is good but,
  - it needs a data structure for mapping terms to termIDs
  - this won't fit in memory for big corpora
- Straightforward solution
  - dynamically create dictionaries
  - store the dictionaries with the blocks

# Single-Pass In-Memory Indexing

## SPIMI

- BSBI is good but,
  - it needs a data structure for mapping terms to termIDs
  - this won't fit in memory for big corpora
- Straightforward solution
  - dynamically create dictionaries
  - store the dictionaries with the blocks

# Single-Pass In-Memory Indexing

```
SPIMI-Invert(tokenStream)
 1  outputFile ← NewFile()
 2  dictionary ← NewHash()
 3  while (free memory available)
 4      do token ← next(tokenStream)
 5          if term(token) ∉ dictionary
 6              then postingsList ← AddToDictionary(dictionary, term(token))
 7              else  postingsList ← GetPostingsList(dictionary, term(token))
 8          if full(postingsList)
 9              then postingsList ← DoublePostingsList(dictionary, term(token))
10          AddToPostingsList(postingsList, docID(token))
11  sortedTerms ← SortTerms(dictionary)
12  WriteBlockToDisk(sortedTerms, dictionary, outputFile)
13  return outputFile
```

# Single-Pass In-Memory Indexing

- So what is different here?
  - SPIMI adds postings directly to a posting list.
    - BSBI first collected (TermID,DocID pairs)
      - then sorted them
      - then aggregated the postings
  - Each posting list is dynamic so there is no posting list sorting
  - Saves memory because a term is only stored once
  - Complexity is more like O(T)
  - Compression enables bigger effective blocks

# Large Scale Indexing

- Key decision in block merge indexing is block size
- In practice, spidering often interlaced with indexing
- Spidering bottlenecked by WAN speed and other factors