Web search

- The original large data problem
- Problem: given a query term provide the list of documents that contain most relevant information on that
- Three main components of web search:
  - Crawling: gathering of web content
  - Indexing: constructing an inverted index
  - Retrieval: rank documents given a query term
Web crawling

- Acquire document collection which form the knowledge database
- In theory: simple program that follows every link and downloads every webpage found
- Challenges of web crawling
  - Overload web servers: wait a fixed amount of time before subsequent requests to the same server to avoid overload
  - Retrieval frequency: webcontent changes at different pace (e.g. news site vs. my groups webpages)
  - Duplicate content: dealing with mirrors etc.
  - Multilingual support: web content is published in many different languages

Inverted index

- Inverted index: An index data structure storing a mapping from content to its locations in the original document
- Necessary for fast full text searches
- Requires significant processing when a document is added to the database
- Popular data structure used in document retrieval systems
- Used on a large scale in search engines
Example

- Two input files:
  Input.txt: The sample input file contains sample keywords.
  Input2.txt: Another input contains different keywords.
- Simple inverted index:
  Another input2.txt
  The input.txt contains input2.txt
  different input2.txt
  file input.txt
  input input.txt input2.txt
  keywords input.txt input2.txt
  sample input.txt

MapReduce it?

- The indexing problem
  - Scalability is critical
  - Must be relatively fast, but need not be real time
  - Fundamentally a batch operation
  - Incremental updates may or may not be important

- The retrieval problem
  - Must have sub-second response time
  - For the web, only need relatively few results
MapReduce: Index Construction

- Map over all documents, assuming one document as input to a mapper
  - Emit term as key, (documentId, termfrequency) as value
  - Emit other information as necessary (e.g., term position)
  - Mapper needs to be able to store all terms related to a document in memory!
- Sort/shuffle: group postings by term
- Reduce
  - Gather and sort the postings (e.g., by documentId or tf)
  - Write postings to disk
- Fundamentally, a large sorting problem

Inverted Indexing: Pseudo-Code

1: \textbf{procedure} Map(a, d)
2: \hspace{1em} \textbf{initialize.associate.array}(H)
3: \hspace{1em} \textbf{for all} \ t \in d \ \textbf{do}
4: \hspace{2em} H[t] \leftarrow H[t] + 1
5: \hspace{1em} \textbf{for all} \ t \in H \ \textbf{do}
6: \hspace{2em} \textbf{emit}(t, \langle a, H[t] \rangle)

1: \textbf{procedure} Reduce(t, \langle a_1, f_1 \rangle, \langle a_2, f_2 \rangle \ldots)
2: \hspace{1em} \textbf{initialize.list}(P)
3: \hspace{1em} \textbf{for all} \ \langle a, f \rangle \in \langle a_1, f_1 \rangle, \langle a_2, f_2 \rangle \ldots \ \textbf{do}
4: \hspace{2em} \textbf{append}(P, \langle a, f \rangle)
5: \hspace{1em} \textbf{sort}(P)
6: \hspace{1em} \textbf{emit}(t, P)
Inverted Indexing: Pseudo-Code

Initial implementation: terms as keys, postings as values
- Reducers must buffer all postings associated with key (to sort)
- Can run out of memory to buffer postings

Another Try...

- Emit key: (t, docid) value: tf
- How is this different?
  - Let the framework do the sorting
  - Term frequency implicitly stored
  - Directly write postings to disk!
Inverted Index: second approach

- Assuming line-by-line input to mappers:
  - Similarly to word count, map emits term as key and value being a tuple of (1, documentId)
  - Requires retrieval of input file name/document Id!
    - Remember, that Hadoop takes all files in a directories and combines them logically before the split
  - Reducer calculates terms frequency per document Id and emits
    Term, documentId, frequency
- Second MapReduce job combines all occurrences of a term to a single line entry

Encoding the Index

- Naïve implementation of an index can be as large as the document itself
- Index compression utilized by search engines

**Conceptually:**

```
fish  1  2  9  1  21  3  34  1  35  2  80  3  ...
```

**In Practice:**

- Don’t encode documentId, encode gaps (or d-gaps)
- Requires index to be sorted by documentID per term

```
fish  1  2  8  1  12  3  13  1  1  2  45  3  ...
```
Overview of Index Compression

- Non-parameterized
  - Unary codes
  - $\gamma$ codes
  - $\delta$ codes
- Parameterized
  - Golomb codes (local Bernoulli model)

Want more detail? Read *Managing Gigabytes* by Witten, Moffat, and Bell!

Unary Codes

- $x \geq 1$ is coded as $x-1$ one bits, followed by 1 zero bit
  - $3 = 110$
  - $4 = 1110$
- Great for small numbers... horrible for large numbers
  - Overly-biased for very small gaps
\( \gamma \) codes

- \( x \geq 1 \) is coded in two parts: length and offset
  - Length \( N = \lceil \log x \rceil + 1 \) (highest power of 2 it contains)
  - Offset: remainder bits in binary encoded in \( N-1 \) bits
  - Length encoded in unary code
  - Concatenate length + offset codes
- Example: 9 in binary is 1001
  - Length \( N = 3+1=4 \), in unary code = 1110
  - Offset = 001 (encoded in \( N-1 = 3 \) bits)
  - \( \gamma \) code = 1110:001

Retrieval in a Nutshell

- Look up postings lists corresponding to query terms
- Traverse postings for each query term
- Store partial query-document scores in accumulators
- Select top \( k \) results to return
Retrieval: Query-At-A-Time

- Evaluate documents one query at a time
  - Usually, starting from most rare term (often with tf-scored postings)

\[
\text{Score}_{[q \to \cdot]}(\text{doc } n) = s
\]

- Tradeoffs
  - Early termination heuristics (good)
  - Large memory footprint (bad), but filtering heuristics possible

\[
\begin{align*}
\text{blue} & : 9 & 2 & 21 & 1 & 35 & 1 & \ldots \\
\text{fish} & : 1 & 2 & 9 & 1 & 21 & 3 & 34 & 1 & 35 & 2 & 80 & 3 & \ldots
\end{align*}
\]

Retrieval: Document-at-a-Time

- Evaluate documents one at a time (score all query terms)

- Tradeoffs
  - Small memory footprint (good)
  - Must read through all postings (bad), but skipping possible
  - More disk seeks (bad), but blocking possible

\[
\begin{align*}
\text{blue} & : 9 & 2 & 21 & 1 & 35 & 1 & \ldots \\
\text{fish} & : 1 & 2 & 9 & 1 & 21 & 3 & 34 & 1 & 35 & 2 & 80 & 3 & \ldots
\end{align*}
\]

Document score in top k?
- Yes: Insert document score, extract-min if queue too large
- No: Do nothing
Retrieval with MapReduce?

- MapReduce is fundamentally batch-oriented
  - Optimized for throughput, not latency
  - Startup of mappers and reducers is expensive
- MapReduce is not suitable for real-time queries!
  - Use separate infrastructure for retrieval...

Important Ideas

- Partitioning (for scalability)
- Replication (for redundancy)
- Caching (for speed)
- Routing (for load balancing)
Term vs. Document Partitioning

Parallel Queries: Map

blue

fish

key = 1, value = { 9:2, 21:1, 35:1 }
Parallel Queries: Reduce

key = 1, value = { 9:2, 21:1, 35:1 }
key = 1, value = { 1:2, 9:1, 21:3, 34:1, 35:2, 80:3 }

Reducer

Element-wise sum of associative arrays

key = 1, value = { 1:2, 9:3, 21:4, 34:1, 35:3, 80:3 }

Sort accumulators to generate final ranking

Query: “blue fish”
doc 21, score=4
doc 2, score=3
doc 35, score=3
doc 80, score=3
doc 1, score=2
doc 34, score=1

3rd Homework

• Rules
  - Each student should deliver
    • Source code (.java files)
    • Documentation (.pdf, .doc, .tex or .txt file)
      - explanations to the code
      - answers to questions
    - Deliver electronically to gabriel@cs.uh.edu
  - Expected by Friday, April 24, 11.59pm
  - In case of questions: ask, ask, ask!

UNIVERSITYofHOUSTON
• Given a data set containing multiple thousand books as raw text files. (Book source: www.gutenberg.org). The goal is to build a simple search engine for identifying the most relevant books from the data set given a search term.

• Part 1: implement the source code required to build an inverted index (MapReduce)
  - Format of the output file:
    term1 file1:tf1, file2:tf2, file3:tf3...
    term2 file1:tf1, file2:tf2, file3:tf3...
  - The order of file:tf should be sorted with higher term frequencies listed first
  - No compression
  - MapReduce job will operate on a per line basis -> you have to detect the name of the input file
    FileSplit fileSplit = (FileSplit) context.getInputSplit();
    String fileName = fileSplit.getPath().getName();

• Part 2: build a simple retriever
  - Should be able to handle more than one query term
  - Results should be based on the sum of term frequencies for a given document
  - Output the top 10 results per query
  - Does not have to be MapReduce, but can
  - Does not have to be Java, but can
  - Additional processing after generating the inverted index is ok - you should just describe in your document
• Raw text documents will be in /bigdata-hw3
  - You should of course just use a small subset (e.g. 3-5 documents) for the code development
• Report the time for generating the inverted index
• Report the time spent on average per query (single term, two simultaneous terms, three simultaneous terms)

Suggestions for improvements
• It's up to you what you do beyond the simple solution, but nicer solutions will get more points
  - Removing quotes from words
  - Remove filler words (the, a, an, has, if, and, or, I, you, we, that, they, them, ...)
  - Handling singular vs. plural (not sure how to do it generally)
  - Sorting output e.g. into separate files based on the starting character of the term to speed up
  - ...
• You will *not* be able to do everything, so make a conscious decision what you choose to implement