MapReduce Algorithms

Algorithms for MapReduce

- Sorting
- Searching
- Indexing
- Classification
- Joining
- TF-IDF

MapReduce Jobs

- Tend to be very short, code-wise
  – IdentityReducer is very common
- "Utility" jobs can be composed
- Represent a *data flow*, more so than a procedure

Sort: Inputs

- A set of files, one value per line.
- Mapper key is file name, line number
- Mapper value is the contents of the line

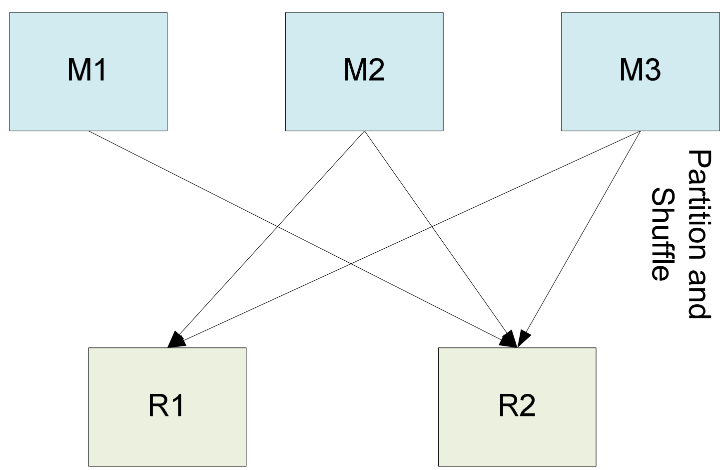
Sort Algorithm

- Takes advantage of reducer properties: (key, value) pairs are processed in order by key; reducers are themselves ordered
- Mapper: Identity function for value

  $(k, v) \rightarrow (v, \_)$
- Reducer: Identity function (k', _) -> (k', "")

Sort: The Trick

- (key, value) pairs from mappers are sent to a particular reducer based on hash(key)
- Must pick the hash function for your data such that $k_1 < k_2 \Rightarrow hash(k_1) < hash(k_2)$



Final Thoughts on Sort

- Used as a test of Hadoop's raw speed
- Essentially "IO drag race"
- Highlights utility of GFS

Search: Inputs

- A set of files containing lines of text
- A search pattern to find

- Mapper key is file name, line number
- Mapper value is the contents of the line
- Search pattern sent as special parameter

Search Algorithm

- Mapper:
  - Given (filename, some text) and “pattern”, if “text” matches “pattern” output (filename, _)
- Reducer:
  - Identity function

Search: An Optimization

- Once a file is found to be interesting, we only need to mark it that way once
- Use *Combiner* function to fold redundant (filename, _) pairs into a single one
  - Reduces network I/O

Indexing: Inputs

- A set of files containing lines of text

- Mapper key is file name, line number
- Mapper value is the contents of the line

Inverted Index Algorithm

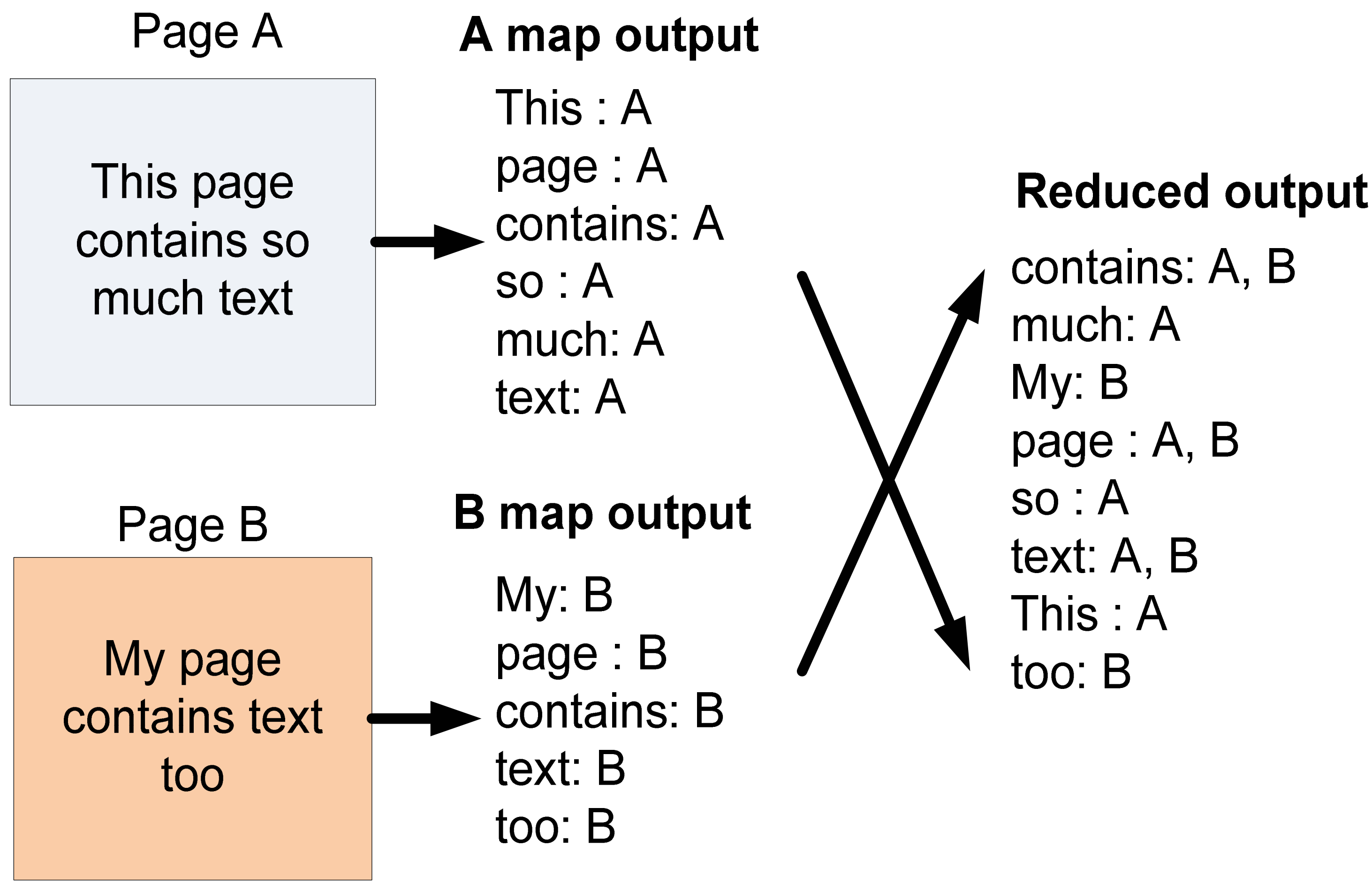
- Mapper: For each word in (file, words), map to (word, file)
- Reducer: Identity function

Index: MapReduce

```
map(pageName, pageText):
    foreach word w in pageText:
      emitIntermediate(w, pageName);
    done

reduce(word, values):
    foreach pageName in values:
      AddToOutputList(pageName);
    done
    emitFinal(FormattedPageListForWord);
```

Index: Data Flow



An Aside: Word Count

- Word count was described in module I
- Mapper for Word Count is (word, 1) for each word in input line
  - Strikingly similar to inverted index
  - Common theme: reuse/modify existing mappers

Bayesian Classification

- Files containing classification instances are sent to mappers
- Map (filename, instance) → (instance, class)
- Identity Reducer

Bayesian Classification

- Existing toolsets exist to perform Bayes classification on instance
  - E.g., WEKA, already in Java!
- Another example of discarding input key

Joining

- Common problem: Have two data types, one includes references to elements of the other; would like to incorporate data by value, not by reference

- Solution: MapReduce Join Pass

Join Mapper

- Read in all values of joiner, joinee classes
- Emit to reducer based on primary key of joinee (i.e., the reference in the joiner, or the joinee’s identity)

Join Reducer

- Joinee objects are emitted as-is
- Joiner objects have additional fields populated by Joinee which comes to the same reducer as them.
  - Must do a secondary sort in the reducer to read the joinee before emitting any objects which join on to it

TF-IDF

- Term Frequency – Inverse Document Frequency
  - Relevant to text processing
  - Common web analysis algorithm

The Algorithm, Formally

$$\mathrm{tf_i} = \frac{n_i}{\sum_k n_k}$$

$$\mathrm{idf_i} = \log \frac{|D|}{|\{d : t_i \in d\}|}$$

$$\mathrm{tfidf} = \mathrm{tf} \cdot \mathrm{idf}$$

- $|D|$: total number of documents in the corpus
- $|\{d : t_i \in d\}|$: number of documents where the term t_i appears (that is $n_i \neq 0$).

Information We Need

- Number of times term X appears in a given document
- Number of terms in each document
- Number of documents X appears in
- Total number of documents

Job 1: Word Frequency in Doc

- Mapper
  - Input: (docname, contents)
  - Output: ((word, docname), 1)
- Reducer
  - Sums counts for word in document
  - Outputs ((word, docname), *n*)
- Combiner is same as Reducer

Job 2: Word Counts For Docs

- Mapper
  - Input: ((word, docname), *n*)
  - Output: (docname, (word, *n*))
- Reducer
  - Sums frequency of individual *n*'s in same doc
  - Feeds original data through
  - Outputs ((word, docname), (*n*, *N*))

Job 3: Word Frequency In Corpus

- Mapper
  - Input: ((word, docname), (*n, N)*)
  - Output: (word, (docname, *n*, *N*, 1))
- Reducer
  - Sums counts for word in corpus
  - Outputs ((word, docname), (*n*, *N*, *m*))

Job 4: Calculate TF-IDF

- Mapper
  - Input: ((word, docname), (n, N, m))
  - Assume D is known (or, easy MR to find it)
  - Output ((word, docname), TF*IDF)
- Reducer
  - Just the identity function

Working At Scale

- Buffering (doc, *n*, *N*) counts while summing 1’s into *m* may not fit in memory
  - How many documents does the word “the” occur in?
- Possible solutions
  - Ignore very-high-frequency words
  - Write out intermediate data to a file
  - Use another MR pass

Final Thoughts on TF-IDF

- Several small jobs add up to full algorithm
- Lots of code reuse possible
  - Stock classes exist for aggregation, identity
- Jobs 3 and 4 can really be done at once in same reducer, saving a write/read cycle
- Very easy to handle medium-large scale, but must take care to ensure flat memory usage for largest scale

Conclusions

- Lots of high level algorithms
- Lots of deep connections to low-level systems

cloudera