

Eddie Davis

CS 530

PA3: Lab Report (11/18/2016)

1 Algorithm Description

Massive scale text searching is one of the quintessential problems in the big data paradigm. It is characterized by short queries (e.g., a web search for "funny cat memes"), and that most of the source documents will not match those queries (i.e., matches are sparse). Such searches are greatly accelerated by maintaining indexes. An inverted index is one in which the search words are the keys, and the values are identifiers for documents in the collection that contain those words. A document can be any body of text, either web page, book, paper, or other. Subsequent searches need only query the index rather than searching the entire collection.

The process for building an inverted index for a collection consists of parsing the documents, splitting on word boundaries (e.g., whitespace, punctuation, etc.), and building a lookup table where each word is linked to the set of documents wherein it can be found. A naive approach might be to parse each file by line, tokenize each line, and inserting the file name into an associative array of lists with tokens as keys. See Figure 1 for an inverted index example, and Algorithm 1 for a potential sequential implementation.

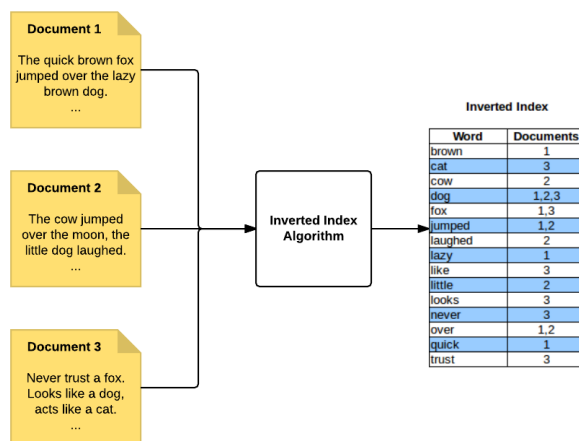


Figure 1: Inverted index example for three sample documents.

Algorithm 1 Sequential Inverted Index

```
procedure SEQUENTIALINVERTEDINDEX(coll) ▷ Collection coll
  M ← new AssociativeArray
  for all doc ∈ coll do ▷ Document doc
    for all word ∈ doc do
      if word ∉ M then
        M{word} ← new Set
        M{word}.add(doc.id)
  F ← new File
  for all word ∈ M do
    F.write(F, M{word})
```

2 Parallelization Approach

MapReduce is a parallel programming model inspired by the **map** and **reduce** statements from functional programming languages. Problems are represented as sets of key-value pairs. A mapper component filters or sorts the data into a common key, and the reducer performs some summarizing operation on all the values containing that key. The Hadoop infrastructure allows these operations to be distributed across a large cluster in a massively parallel fashion.

The inverted index problem is a natural fit for the MapReduce paradigm, particularly for large sets of documents as a sequential search would be quite time intensive. The initial MapReduce implementation is described in Algorithm 2.

Algorithm 2 MapReduce Inverted Index 1

```
procedure SEQUENTIALINVERTEDINDEX(coll) ▷ Collection coll
  M ← new AssociativeArray
  for all doc ∈ coll do ▷ Document doc
    for all word ∈ doc do
      if word ∉ M then
        M{word} ← new Set
        M{word}.add(doc.id)
  F ← new File
  for all word ∈ M do
    F.write(F, M{word})
```

3 Experimental Setup

The run times of the sequential version of the program (**sequential-bucketsort**) are the baseline (control) for this experiment, or the $T_s(n)$ for the speedup calculations.

The experimental groups will run the parallel version (`parallel-bucketsort`) with processes $p \in (2, 4, 8, 16)$, and iterations $n \in (1\text{E}9, 2\text{E}9, 5\text{E}9)$. In order to ensure that the code performs well on different data distributions, three random seed values $s \in \{6764, 445455, 2876533\}$ are included, with the mean run time reported.

The experiments were performed on the **onyx** Fedora Linux cluster. Each node contains an Intel(R) Xeon(R) E31225 processor with four cores clocked at 3.10 GHz. The CPUs include 6144 KB of level three (L3) cache shared by the four cores, 256 KB of level two (L2) cache, and 64 KB of level one (L1) cache per node. Each node has 8 GB of main memory (DRAM) onboard.

4 Experimental Results

5 Conclusions