MapReduce & Distributed Indexing

Acknowledgments: Many slides in this lecture were adapted from Jimmy Lin (Waterloo)
Documents are parsed to extract words and these are saved with the Document ID.

I did enact Julius Caesar. I was killed i' the Capitol; Brutus killed me.

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious.
Simple Index building alg.

procedure BUILDINDEX(D)
    I ← HashTable()
    n ← 0
    for all documents $d \in D$ do
        $n \leftarrow n + 1$
        $T \leftarrow \text{Parse}(d)$
        Remove duplicates from $T$
        for all tokens $t \in T$ do
            if $d \notin I$ then
                $I_d \leftarrow \text{Array}()$
                end if
                $I_d$.append($n$)
            end for
        end for
    end for
    return $I$
end procedure

-$D$ is a set of text documents
-$I$ is an inverted list storage
-$n$ is the document numbering

-$T$ is the parse document into tokens

Problems:
- Dict must be stored in memory
- Sequential. How to parallelize?
Runtime Analysis

• 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 \( \log_2(N) \) comparisons?

→ > 300 days!
Runtime Analysis (cont’d)

• 100,000,000 records
• $N \log_2 N$ is $= 2,657,542,475.91$ comparisons
• 2 disk seeks per comparison
  
  $= 13,287,712.38$ seconds $\times 2$

  $= 26,575,424.76$ seconds

  $= 442,923.75$ minutes

  $= 7,382.06$ hours

  $= 307.59$ days
Scaling index construction

- In-memory index construction does not scale.
- How can we construct an index for very large collections?
- Taking into account the hardware constraints we just learned about . . .
- Memory, disk, speed, etc.
Sort-based index construction

- As we build the index, we parse docs one at a time.
  – While building the index, we cannot easily exploit compression tricks (you can, but much more complex)
- The final postings for any term are incomplete until the end.
- At 12 bytes per non-positional postings entry \((term, doc, freq)\), demands a lot of space for large collections.
- \(T = 100,000,000\) in the case of RCV1
  – So ... we can do this in memory in 2009, but typical collections are much larger. E.g. the New York Times provides an index of >150 years of newswire
- Thus: We need to store intermediate results on disk.
RCV1 and other “Toy” Datasets

- 800,000 Documents $\rightarrow$ $\sim$ 1Gb of data
- NYT 150 years archive $\rightarrow$ $\sim$ 3 Gb of data
- Wikipedia (English): $\sim$3M docs $\rightarrow$ $\sim$ 5 Gb of data
How much data?

Google

- Hadoop: 10K nodes, 150K cores, 150 PB (2008)
- Crawls 20B web pages a day (2012)
- Search index is 100+ PB (5/2014)
- Bigtable serves 2+ EB, 600M QPS (3/2014)

Yahoo!


eBay

- Processes 20 PB a day (2008)
- Crawls 20B web pages a day (2012)
- Search index is 100+ PB (5/2014)
- Bigtable serves 2+ EB, 600M QPS (3/2014)

Facebook

- 150 PB on 50k+ servers running 15k apps (6/2011)
- Search index is 100+ PB (5/2014)

Amazon Web Services

- 300 PB data in Hive + 600 TB/day (4/2014)
- S3: 2T objects, 1.1M request/second (4/2013)

JPMorgan Chase

- 150 PB on 50k+ servers running 15k apps (6/2011)

LHC

- ~15 PB a year

LSST

- 6-10 PB a year (~2020)

SKA

- 0.3 – 1.5 EB per year (~2020)

640K ought to be enough for anybody.
Google data centers

- Data centers come in modules of thousands of machines each.
- Distributed around the world.
- Estimate: > 2.5 million servers, > 10 million processors/cores (2013)
  - Estimate: Google installs 100,000 servers each quarter.
- → 10% of the computing capacity of the world!
Parallelization Challenges

• Individual machines are fault-prone
  – Can unpredictably slow down or fail

• How do we exploit such a pool of machines?
  – How do we assign work units to workers?
  – What if we have more work units than workers?
  – What if workers need to share partial results?
  – How do we aggregate partial results?
  – How do we know all the workers have finished?
  – What if workers die?

What is the common theme of all of these problems?
Typical Large-Data Problem

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Key idea: provide a functional abstraction for these two operations
MapReduce

- Programmers specify two functions:
  \[ \text{map} \ (k, v) \rightarrow <k', v'>^* \]
  \[ \text{reduce} \ (k', v') \rightarrow <k', v'>^* \]
  - All values with the same key are sent to the same reducer

- The execution framework handles everything else...
Anatomy of Map-Reduce

Shuffle and Sort: aggregate values by keys

map

map

map

map

a

b

c

c

a

b

c

reduce

reduce

reduce

r_1

r_2

r_3
MapReduce “Runtime”

• Handles scheduling
  – Assigns workers to map and reduce tasks
• Handles “data distribution”
  – Moves processes to data
• Handles synchronization
  – Gathers, sorts, and shuffles intermediate data
• Handles errors and faults
  – Detects worker failures and restarts
• Everything happens on top of a distributed FS (later)
MapReduce Programming

- Programmers specify two functions:
  - **map** \((k, v) \rightarrow <k', v'>\)*
  - **reduce** \((k', v') \rightarrow <k', v'>\)*
    - All values with the same key are reduced together
- The execution framework handles everything else...
- Not quite... usually, programmers also specify:
  - **partition** \((k', \text{number of partitions}) \rightarrow \text{partition for } k'\)
    - Often a simple hash of the key, e.g., hash\(k'\) \mod n
    - Divides up key space for parallel reduce operations
  - **combine** \((k', v') \rightarrow <k', v'>\)*
    - Mini-reducers that run in memory after the map phase
    - Used as an optimization to reduce network traffic
Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)

Map Reduce Anatomy (2)
“Hello World”: Word Count

Map(String docid, String text):
  for each word w in text:
    Emit(w, 1);

Reduce(String term, Iterator<Int> values):
  int sum = 0;
  for each v in values:
    sum += v;
    Emit(term, value);
Map-Reduce example 2: credit card totals

Map

procedure MapCreditCards(input)
    while not input.done() do
        record ← input.next()
        card ← record.card
        amount ← record.amount
        Emit(card, amount)
    end while
end procedure

Reduce

procedure ReduceCreditCards(key, values)
    total ← 0
    card ← key
    while not values.done() do
        amount ← values.next()
        total ← total + amount
    end while
    Emit(card, total)
end procedure
MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python

- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, now an Apache project
  - Used at Yahoo, Facebook, Twitter, LinkedIn, Netflix, ...
  - The *de facto* big data processing platform
  - Large and expanding software ecosystem

- Lots of custom research implementations
  - For GPUs, cell processors, etc.
Solr, ElasticSearch: Lucene on Hadoop

- [https://www.elastic.co/products/elasticsearch](https://www.elastic.co/products/elasticsearch)
Basic Hadoop API*

- **Mapper**
  - `void setup(Mapper.Context context)`
    Called once at the beginning of the task
  - `void map(K key, V value, Mapper.Context context)`
    Called once for each key/value pair in the input split
  - `void cleanup(Mapper.Context context)`
    Called once at the end of the task

- **Reducer/Combiner**
  - `void setup(Reducer.Context context)`
    Called once at the start of the task
  - `void reduce(K key, Iterable<V> values, Reducer.Context context)`
    Called once for each key
  - `void cleanup(Reducer.Context context)`
    Called once at the end of the task

*Note that there are two versions of the API!
Basic Hadoop API*

• Partitioner
  – int getPartition(K key, V value, int numPartitions)
    Get the partition number given total number of partitions

• Job
  – Represents a packaged Hadoop job for submission to cluster
  – Need to specify input and output paths
  – Need to specify input and output formats
  – Need to specify mapper, reducer, combiner, partitioner classes
  – Need to specify intermediate/final key/value classes
  – Need to specify number of reducers (but not mappers, why?)

*Note that there are two versions of the API!
Data Types in Hadoop: Keys and Values

**Writable**
- Defines a de/serialization protocol.
- Every data type in Hadoop is a Writable.

**WritableComprable**
- Defines a sort order. All keys must be of this type (but not values).

**Concrete classes for different data types**
- IntWritable
- LongWritable
- Text
- ...

**SequenceFiles**
- Binary encoded of a sequence of key/value pairs
“Hello World”: Word Count

Map(String docid, String text):
    for each word w in text:
        Emit(w, 1);

Reduce(String term, Iterator<Integer> values):
    int sum = 0;
    for each v in values:
        sum += v;
        Emit(term, value);
private static class MyMapper
  extends Mapper<LongWritable, Text, Text, IntWritable> {

  private final static IntWritable ONE = new IntWritable(1);
  private final static Text WORD = new Text();

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    String line = ((Text) value).toString();
    StringTokenizer itr = new StringTokenizer(line);
    while (itr.hasMoreTokens()) {
      WORD.set(itr.nextToken());
      context.write(WORD, ONE);
    }
  }
}
private static class MyReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

    private final static IntWritable SUM = new IntWritable();

    @Override
    public void reduce(Text key, Iterable<IntWritable> values,
            Context context) throws IOException,
            InterruptedException {
        Iterator<IntWritable> iter = values.iterator();
        int sum = 0;
        while (iter.hasNext()) {
            sum += iter.next().get();
        }
        SUM.set(sum);
        context.write(key, SUM);
    }
}
Three Gotchas

• Avoid object creation if possible
  – Reuse Writable objects, change the payload
• Execution framework reuses value object in reducer
• Passing parameters via class statics
Getting Data to Mappers and Reducers

• Configuration parameters
  – Directly in the Job object for parameters

• “Side data”
  – DistributedCache
  – Mappers/reducers read from HDFS in setup method
Complex Data Types in Hadoop

• How do you implement complex data types?
  • The easiest way:
    – Encoded it as Text, e.g., (a, b) = “a:b”
    – Use regular expressions to parse and extract data
    – Works, but pretty hack-ish
  • The hard way:
    – Define a custom implementation of Writable(Comparable)
    – Must implement: readFields, write, (compareTo)
    – Computationally efficient, but slow for rapid prototyping
    – Implement WritableComparator hook for performance
  • Somewhere in the middle:
    – Cloud⁹ (via lin.tl) offers JSON support and lots of useful Hadoop types
    – Quick tour...
Anatomy of a Job

• MapReduce program in Hadoop = Hadoop job
  – Jobs are divided into map and reduce tasks
  – An instance of running a task is called a task attempt (occupies a slot)
  – Multiple jobs can be composed into a workflow

• Job submission:
  – Client (i.e., driver program) creates a job, configures it, and submits it to jobtracker
  – That’s it! The Hadoop cluster takes over...
Anatomy of a Job

• Behind the scenes:
  – Input splits are computed (on client end)
  – Job data (jar, configuration XML) are sent to JobTracker
  – JobTracker puts job data in shared location, enqueues tasks
  – TaskTrackers poll for tasks
  – Off to the races...
InputFile

InputFormat

InputSplit

RecordReader

Mapper

Intermediates

Source: redrawn from a slide by Cloudera, cc-licensed
Input and Output

• InputFormat:
  – TextInputFormat
  – KeyValueTextInputFormat
  – SequenceFileInputFormat
  – ...

• OutputFormat:
  – TextOutputFormat
  – SequenceFileOutputFormat
  – ...

Shuffle and Sort in Hadoop

• Probably the most complex aspect of MapReduce

• Map side
  – Map outputs are buffered in memory in a circular buffer
  – When buffer reaches threshold, contents are “spilled” to disk
  – Spills merged in a single, partitioned file (sorted within each partition): combiner runs during the merges

• Reduce side
  – First, map outputs are copied over to reducer machine
  – “Sort” is a multi-pass merge of map outputs (happens in memory and on disk): combiner runs during the merges
  – Final merge pass goes directly into reducer
Shuffle and Sort

Map

Reducer

Combiner

other mappers

other reducers

intermediate files (on disk)

merged spills (on disk)

spills (on disk)

circular buffer (in memory)
How do we get data to the workers?

What’s the problem here?
Distributed File System

- Don’t move data to workers... move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local

- Why?
  - (Perhaps) not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable

- A distributed file system is the answer
  - GFS (Google File System) for Google’s MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop
HDFS Architecture

- Application
  - HDFS Client
    - (file name, block id)
      - (block id, block location)
      - (block id, byte range)
        - block data
  - HDFS datanode
    - Linux file system
      - instructions to datanode
      - datanode state
      - block data
      - block 3df2

- HDFS namenode
  - File namespace
    - /foo/bar
“On the first day of logging the Facebook clickstream, more than 400 gigabytes of data was collected. The load, index, and aggregation processes for this data set really taxed the Oracle data warehouse. Even after significant tuning, we were unable to aggregate a day of clickstream data in less than 24 hours.”

Jeff Hammerbacher, Information Platforms and the Rise of the Data Scientist.
In, Beautiful Data, O’Reilly, 2009.
Logging/Processing Bottleneck

• Log processing is typically a nightly task:
  – What happens if processing 24 hours of data takes longer than 24 hours?

• Hadoop is perfect:
  – Ingest is limited by speed of HDFS
  – Scales out with more nodes
  – Massively parallel
  – Ability to use any processing tool
  – Cheaper than parallel databases
  – Batch process anyway!
Twitter Logging Architecture

Diagram showing the architecture of Twitter logging with components such as MySQL, Vertica, Hadoop, Scribe, Real-Time Aggs, Analytics, BI, Data-Powered Products, Dashboards, Prod NoSQL stores, and Ad-Hoc Analysis.
Twitter Internal Support

circa ~2010
~150 people total
~60 Hadoop nodes
~6 people use analytics stack daily

circa ~2012
~1400 people total
10s of Ks of Hadoop nodes, multiple DCs
10s of PBs total Hadoop DW capacity
~100 TB ingest daily
dozens of teams use Hadoop daily
10s of Ks of Hadoop jobs daily
Recall IIR 1 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.
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
  – For the web, crawling is a challenge in itself

• The retrieval problem
  – Must have sub-second response time
  – For the web, only need relatively few results

Perfect for MapReduce!

... not so good...
MapReduce: Index Construction

- Map over all documents
  - Emit term as key, (docno, tf) as value
  - Emit other information as necessary (e.g., term position)
- Sort/shuffle: group postings by term
- Reduce
  - Gather and sort the postings (e.g., by docno or tf)
  - Write postings to disk
- MapReduce does all the heavy lifting!
Inverted Indexing with MapReduce

Map

Reduce

Shuffle and Sort: aggregate values by keys
Inverted Indexing: Pseudo-Code

1: class Mapper
2:    method MAP(docid n, doc d)
3:        \(H \leftarrow \text{new AssociativeArray}\)  \(\triangleright \) histogram to hold term frequencies
4:        for all term \(t \in \text{doc d}\) do  \(\triangleright \) processes the doc, e.g., tokenization and stopword removal
5:            \(H\{t\} \leftarrow H\{t\} + 1\)
6:    for all term \(t \in H\) do
7:        EMIT(term \(t\), posting \(\langle n, H\{t\}\rangle\))  \(\triangleright \) emits individual postings

1: class Reducer
2:    method REDUCE(term \(t\), postings \([\langle n_1, f_1\rangle \ldots]\))
3:        \(P \leftarrow \text{new List}\)
4:        for all \(\langle n, f \rangle \in \text{postings} \,[\langle n_1, f_1\rangle \ldots]\) do
5:            \(P.\text{APPEND}((\langle n, f \rangle))\)  \(\triangleright \) appends postings unsorted
6:        \(P.\text{SORT}()\)  \(\triangleright \) sorts for compression
7:        EMIT(term \(t\), postingsList \(P\))
Positional Indexes

Map

Reduce

Shuffle and Sort: aggregate values by keys

Doc 1

Doc 2

Doc 3
Inverted Indexing: Pseudo-Code

1: class Mapper
2:     method MAP(docid n, doc d)
3:         \( H \leftarrow \text{new AssociativeArray} \)  \( \triangleright \) histogram to hold term frequencies
4:         \textbf{for all} term \( t \in \text{doc} \ \text{do} \)  \( \triangleright \) processes the doc, e.g., tokenization and stopword removal
5:         \( H\{t\} \leftarrow H\{t\} + 1 \)
6:         \textbf{for all} term \( t \in H \ \text{do} \)
7:             \text{EMIT}(term \ t, \ \text{posting} \ \langle n, H\{t\} \rangle)  \( \triangleright \) emits individual postings

1: class Reducer
2:     method REDUCE(term \ t, \ postings \ \[ \langle n_1, f_1 \rangle \ldots \])
3:         \( P \leftarrow \text{new List} \)
4:         \textbf{for all} \ \langle n, f \rangle \in \text{postings} \ \[ \langle n_1, f_1 \rangle \ldots \] \ \text{do}
5:             \( P\text{.APPEND}(\langle n, f \rangle) \)
6:             \( P\text{.SORT}() \)  \( \triangleright \) appends postings unsorted
7:             \text{EMIT}(term \ t, \ \text{postingsList} \ P)  \( \triangleright \) sorts for compression

What's the problem?
Scalability Bottleneck

- Initial implementation: terms as keys, postings as values
  - Reducers must buffer all postings associated with key (to sort)
  - What if we run out of memory to buffer postings?
- Uh oh!
Another Try...

How is this different?
• Let the framework do the sorting
• Term frequency implicitly stored
Inverted Indexing: Pseudo-Code

1: class Mapper
2:   method Map(docid n, doc d)
3:     \( H \leftarrow \text{new AssociativeArray} \)
4:     for all term \( t \in \text{doc} \) do
5:         \( H\{t\} \leftarrow H\{t\} + 1 \) \( \triangleright \) builds a histogram of term frequencies
6:     for all term \( t \in H \) do
7:         \( \text{Emit}(\text{tuple } \langle t, n \rangle, \text{tf } H\{t\}) \) \( \triangleright \) emits individual postings, with a tuple as the key

1: class Partitioner
2:   method Partition(two \( \langle t, n \rangle \), \text{tf } f)
3:     return \( \text{Hash}(t) \mod \text{NumOfReducers} \) \( \triangleright \) keys of same term are sent to same reducer

1: class Reducer
2:   method Initialize
3:     \( t_{prev} \leftarrow \emptyset \)
4:     \( P \leftarrow \text{new PostingsList} \)
5:   method Reduce(two \( \langle t, n \rangle \), \text{tf } [f])
6:     if \( t \neq t_{prev} \land t_{prev} \neq \emptyset \) then
7:         \( \text{Emit}(\text{term } t, \text{postings } P) \) \( \triangleright \) emits postings list of term \( t_{prev} \)
8:         \( P.\text{Reset}() \)
9:         \( P.\text{Append}(\langle n, f \rangle) \) \( \triangleright \) appends postings in sorted order
10:        \( t_{prev} \leftarrow t \)
11:   method Close
12:     \( \text{Emit}(\text{term } t, \text{postings } P) \) \( \triangleright \) emits last postings list from this reducer
Inverted Index (Again)

<table>
<thead>
<tr>
<th>Term</th>
<th>Doc 1</th>
<th>Doc 2</th>
<th>Doc 3</th>
<th>Doc 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>cat</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>egg</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>fish</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>green</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>ham</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>hat</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>one</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>red</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>two</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**tf**

**df**

- **Doc 1**: blue, cat, egg, fish, green, one, red, two
- **Doc 2**: blue, cat, egg, fish, green, one, red, two
- **Doc 3**: green, ham, hat, one, red, two
- **Doc 4**: green, ham, hat, one, red, two
We’d like to store the \textit{df} at the front of the postings list.

But we don’t know the \textit{df} until we’ve seen all postings!

Write postings

Sound familiar?
Getting the $df$

- In the mapper:
  - Emit “special” key-value pairs to keep track of $df$

- In the reducer:
  - Make sure “special” key-value pairs come first: process them to determine $df$

- Remember: proper partitioning!
## Getting the $df$: Modified Mapper

**Doc 1**

- **one fish, two fish**

<table>
<thead>
<tr>
<th>(key)</th>
<th>(value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fish 1</td>
<td>[2,4]</td>
</tr>
<tr>
<td>one 1</td>
<td>[1]</td>
</tr>
<tr>
<td>two 1</td>
<td>[3]</td>
</tr>
</tbody>
</table>

Input document…

Emit normal key-value pairs…

**fish** ★ [1]

Emit “special” key-value pairs to keep track of $df$…

**one** ★ [1]

**two** ★ [1]
Getting the \textit{df}: Modified Reducer

First, compute the \textit{df} by summing contributions from all “special” key-value pair...

Important: properly define sort order to make sure “special” key-value pairs come first!

Write postings
Postings Encoding

Conceptually:

fish

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

In Practice:

• Don’t encode docnos, encode gaps (or d-gaps)
• But it’s not obvious that this save space…

fish

1 2 8 1 12 3 13 1 1 2 45 3 ...

3/30/2016
CS 572: Information Retrieval, Spring 2016
Inverted Indexing: IP

1: class Mapper
2:    method Map(docid n, doc d)
3:        H ← new AssociativeArray
4:        for all term t ∈ doc d do
5:            H{t} ← H{t} + 1
6:        for all term t ∈ H do
7:            Emit(tuple ⟨t, n⟩, tf H{t})  # emits individual postings, with a tuple as the key

1: class Partitioner
2:    method Partition(tuple ⟨t, n⟩, tf f)
3:        return Hash(t) mod NumOfReducers  # keys of same term are sent to same reducer

1: class Reducer
2:    method Initialize
3:        t_prev ← ∅
4:        P ← new PostingsList
5:    method Reduce(tuple ⟨t, n⟩, tf [f])
6:        if t ≠ t_prev ∧ t_prev ≠ ∅ then
7:            Emit(term t, postings P)
8:            P.reset()
9:            P.Append(⟨n, f⟩)
10:       t_prev ← t
11:    method Close
12:        Emit(term t, postings P)  # emits last postings list from this reducer

What’s the assumption?
Inverted Indexing: LP

1: class Mapper
2:   method Initialize
3:       \( M \leftarrow \text{new AssociativeArray} \) \( \triangleright \) holds partial lists of postings
4:   method MAP(docid n, doc d)
5:       \( H \leftarrow \text{new AssociativeArray} \) \( \triangleright \) builds a histogram of term frequencies
6:       for all term \( t \in \text{doc d} \) do
7:           \( H\{t\} \leftarrow H\{t\} + 1 \)
8:       for all term \( t \in H \) do
9:           \( M\{t\}.\text{Add}(\text{posting}\ \langle n, H\{t\}\rangle) \)
10:          if MEMORYFULL() then
11:             FLUSH()
12:   method FLUSH \( \triangleright \) flushes partial lists of postings as intermediate output
13:       for all term \( t \in M \) do
14:           \( P \leftarrow \text{SORTANDENCODEPOSTINGS}(M\{t\}) \)
15:           \text{EMIT}(\text{term t, postingsList P})
16:           M.CLEAR()
17:   method CLOSE
18:       FLUSH()
Inverted Indexing: LP

1: class Reducer
2:  method Reduce(term t, postingsLists [P1, P2, ...])
3:      Pf ← new List ▷ temporarily stores partial lists of postings
4:      R ← new List ▷ stores merged partial lists of postings
5:      for all P ∈ postingsLists [P1, P2, ...] do
6:          Pf.Add(P)
7:      if MEMORYNEARLYFULL() then
8:          R.Add(MERGELISTS(Pf))
9:          Pf.Clear()
10:         R.Add(MERGELISTS(Pf))
11:       Emit(term t, postingsList MERGELISTS(R)) ▷ emits fully merged postings list of term t
MapReduce it?

• The indexing problem
  – Scalability is paramount
  – Must be relatively fast, but need not be real time
  – Fundamentally a batch operation
  – Incremental updates may or may not be important
  – For the web, crawling is a challenge in itself

• The retrieval problem
  – Must have sub-second response time
  – For the web, only need relatively few results
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)

The rest is just details!
Given page $x$ with inlinks $t_1...t_n$, where

- $C(t)$ is the out-degree of $t$
- $\alpha$ is probability of random jump
- $N$ is the total number of nodes in the graph

\[
PR(x) = \alpha \left( \frac{1}{N} \right) + (1 - \alpha) \sum_{i=1}^{n} \frac{PR(t_i)}{C(t_i)}
\]
Computing PageRank

- Properties of PageRank
  - Can be computed iteratively
  - Effects at each iteration are local

- Sketch of algorithm:
  - Start with seed $PR_i$ values
  - Each page distributes $PR_i$ “credit” to all pages it links to
  - Each target page adds up “credit” from multiple in-bound links to compute $PR_{i+1}$
  - Iterate until values converge
Simplified PageRank

• First, tackle the simple case:
  – No random jump factor
  – No dangling nodes

• Then, factor in these complexities...
  – Why do we need the random jump?
  – Where do dangling nodes come from?
Sample PageRank Iteration (1)
Sample PageRank Iteration (2)
PageRank in MapReduce

Diagram showing the relationship between nodes n1, n2, n3, n4, and n5 in a MapReduce context.
PageRank Pseudo-Code

1: class Mapper
2:   method MAP(nid n, node N)
3:     p ← N.PAGERANK/|N.ADJACENCYLIST|  ▶ Pass along graph structure
4:     EMIT(nid n, N)  ▶ Pass PageRank mass to neighbors
5:   for all nodeid m ∈ N.ADJACENCYLIST do
6:     EMIT(nid m, p)

1: class Reducer
2:   method REDUCE(nid m, [p_1, p_2, ...])
3:     M ← ∅  ▶ Recover graph structure
4:     for all p ∈ counts [p_1, p_2, ...] do
5:       if ISNODE(p) then
6:         M ← p
7:       else
8:         s ← s + p  ▶ Sums incoming PageRank contributions
9:     M.PAGERANK ← s
10:    EMIT(nid m, node M)
Resources for today’s lecture

- Original publication on MapReduce: Dean and Ghemawat (2004)
- Jimmy Lin’s Hadoop tutorial: 