Parallelism for IR

CISC689/489-010, Lecture #19
Monday, April 27th
Ben Carterette

IR in One Slide

• Document processing and indexing:
  – Each document turned into a vector of features
  – Vectors of features added to inverted list
  – Inverted lists stored on disk
• Query processing:
  – Inverted lists retrieved from disk
  – Each list decompressed and processed sequentially
  – Scores accumulated in array
• I see a lot of room for parallelization...
Parallelization

• Basic idea: we have a machine with multiple CPUs, or many machines connected together via a network
  — ir.cis has 8 nodes in a network; each node has 2 CPUs with 4 cores; 64 cores total
• We have a task that can be divided into smaller subtasks
• Parallelization involves:
  — Dividing task into smaller subtasks
  — Sending subtasks to nodes for processing
  — Aggregating results from the nodes

Simple Parallel Query Processing

• Many users submitting queries at the same time
• We don’t want users to have to wait on each other
• Idea: replicate the index on each node
  — With n nodes, n users can submit queries simultaneously
  — Space usage is very high: n*size of index
Distributed Query Processing

• Instead, split index up across nodes
• Basic process
  – All queries sent to a director machine
  – Director then sends messages to many index nodes
  – Each index node does some portion of the query processing
  – Director organizes the results and returns them to the user
• Two main approaches
  – Document distribution
    • by far the most popular
  – Term distribution

Distributed Query Processing

• Document distribution
  – each index server acts as a search engine for a small fraction of the total collection
  – director sends a copy of the query to each of the index servers, each of which returns the top-\(k\) results
  – results are merged into a single ranked list by the director
• Collection statistics should be shared for effective ranking
Distributed Query Processing

• Parallel document-at-a-time processing:
  — For each document D
    • Locate node that indexed D
    • That node calculates a score for D
    • Each node returns a ranked list of scores for its own documents
  — Director node merges scores

Advantages of Document Partitioning

• Each node has accumulators only for its own documents
  — Lower memory usage
  — Less data transferred across network
• Each node can use document-at-a-time optimizations
  — Score just the top k documents for faster processing and better resource use
Distributed Query Processing

• Term distribution
  – Single index is built for the whole cluster of machines
  – Each inverted list in that index is then assigned to one index server
    • in most cases the data to process a query is not stored on a single machine
  – One of the index servers is chosen to process the query
    • usually the one holding the longest inverted list
  – Other index servers send information to that server
  – Final results sent to director

Distributed Query Processing

• Parallel term-at-a-time processing:
  – Locate node n1 that has longest inverted list
  – For each term t
    • Locate node that has list for t
    • Direct that node to send its list to n1
    • n1 processes list and accumulates document scores
  – n1 returns final scores to director
Disadvantages of Term Partitioning

- Term inverted lists can get very long, which means a lot of data transfer in the network
- Very common query terms will result in more load on the nodes that contain them
- Less ability to optimize

Fault Tolerance and Redundancy

- Nodes will fail
  - If failure probability is p, and there are n nodes, probability that at least one is down is $1 - (1 - p)^n$
- For strict partitioning, a failed node means the best result may not be found
  - Term partitioning: some inverted lists cannot be found
    - Probability that m-term query can be processed = $\binom{n-1}{m}$
  - Document partitioning: some documents cannot be scored
    - Probability that j out of top k results will be missed = $\binom{k}{j} \left( \frac{1}{n} \right)^j \left( \frac{n-1}{n} \right)^{k-j}$
Fault Probability Example

- n = 64 nodes, probability of failure is 0.01
- Probability at least one node is down = 47%
- With term partitioning:
  - If one node is down, probability that 3-term query cannot be processed is $1 - (63/64)^3 = 4.6\%$
  - If two nodes down, probability is 9.1%
  - Probability increases with query length and failed nodes
- With document partitioning:
  - If one node is down, probability that one of the top 10 results will be lost is $10 \times (1/64) \times (63/64)^9 = 13\%$
  - If two nodes down, probability is 23%
  - Probability increases with failed nodes, decreases with number of results possibly lost

Redundancy

- Replicate indexes to handle faults
- Each partition stored on two different nodes
  - Load on that partition distributed evenly between the nodes
- If one fails, all of its load is redirected to the duplicate
Parallel Indexing

- Very simple indexing pseudocode:
  - Index(C)
    - For each document D
      - For each term t
        » Find inverted list for t (create it if it doesn’t exist)
        » Append D to the end of the list
  - How can we parallelize it?

Two Approaches

- Term partitioning
  - Index(C)
    - For each document D in C
      - For each term t in D
        » ProcessAtNode(t, D)
  - ProcessAtNode(t, D)
    - Find inverted list for t (create it if it doesn’t exist)
    - Append D to the end of the list
- Document partitioning or no partitioning
  - Index(C)
    - For each document D
      - ProcessAtNode(D)
      - Merge partial inverted lists
  - ProcessAtNode(D)
    - For each term t in D
      - Find inverted list for t in local space (create if it doesn’t exist)
      - Append D to list
MapReduce

• MapReduce is a distributed programming tool for indexing and analysis tasks
• Basic idea comes from Lisp:
  – Map a simple function across a list of items
  – Reduce uses a simple function to combine a list of items into one
• Simple example of map and reduce functions:
  – map(count, (a, b, c, a, d)) \rightarrow (a, 1), (b, 1), (c, 1), (a, 1), (d, 1)
  – reduce(+, (1, 2, 3, 1, 4)) \rightarrow 11

MapReduce Setup

• n map nodes, m reduces nodes
• Engine developer defines a map operator that takes a value and outputs a set of values
• Developer defines a reduce operator that takes a set of values and reduces them to a single value
• Details of distributing jobs across nodes handled inside the MapReduce internals
Map

• Define map in terms of a key and value
  — E.g. key = document name/number, value = contents
• For each value, apply some function
  — E.g. document parser
• The function can then be applied to each value over n nodes
  — Parse the contents of n documents in parallel

Map Pseudocode

Map(String key, String value)
   // key = document name
   // value = document contents
   T = parse(value)
   For each term t in T
      output(t, 1)
Reduce

• Define reduce in terms of a key and set of values
  – E.g. a term and m 1s
• Apply some function to the set of values
  – E.g. sum of m 1s = m
• Return the key and the reduced value
• Many reduce jobs can run in parallel, since they only require access to a key, value pair

Reduce Pseudocode

Reduce(String key, Array values)
    // key = a term
    // values = a list of integers
    m = 0
    For each value v
        m += v
    output(key, m)
Shuffling

- The reducer requires that all pairs with the same key are together
  - E.g. (t1, (1, 1, 1)), (t2, (1, 1)), ...
- The mapper just outputs the key with a 1
- Before applying the reducer, we apply a shuffler to aggregate the map outputs
  - (t1, 1), (t2, 1), (t1, 1), (t1, 1), (t2, 1) ...
  - $\rightarrow$ (t1, (1, 1, 1)), (t2, (1, 1)), ...

Workflow
Partitioning

- Before the shuffler can be applied, it needs to collect all of the map outputs
  - Possibly requiring sending them over the network
- To reduce bandwidth requirements, we could make sure all map outputs end up on the same node that will be reducing them
- Use a hash function to determine which node the map output will go to
  - hash(word) mod n

Parallelized Workflow
Using MapReduce

- How many map jobs?
- How big should each job be?
- How many reduce jobs?

Number and Size of Jobs

- Many more jobs than processors
  - This makes load balancing easier: whenever one node can take more jobs, there will be one available
  - Original paper: 200,000 jobs for 5,000 machines
- Size of jobs
  - As small as possible
  - Original paper: no more than 64Mb of data
  - Smaller jobs are easier to restart if they fail
- Number of reduce nodes
  - Original paper: 5,000 for 200,000 map jobs
Advantages of MapReduce

• Not just for parallelization
  – Also used for processing very large files that could not
    be kept in memory

• Fault tolerance
  – If a worker node fails, the job on it can simply be
    redistributed to another node

• Redundant execution
  – As map jobs are finishing, if one worker is particularly
    slow, redistribute its jobs to finished workers
  – Whoever finishes first “wins”

Indexing Example

```plaintext
procedure MAPDOCUMENTSTOPOSTINGS(input)
  while not input.done() do
    document ← input.next()
    number ← document.number
    position ← 0
    tokens ← Parse(document)
    for each word w in tokens do
      Emit(w, document.position)
      position ← position + 1
    end for
  end while
end procedure

procedure REDUCEPOSTINGSTOListS(key, values)
  word ← key
  WriteWord(word)
  while not input.done() do
    EncodePosting(values.next())
  end while
end procedure
```
Other Applications

• Link extraction and counting
  – Mapper outputs set of (URL, 1) tuples; reducer adds up 1s for each URL
• PageRank (more on Wednesday)
• Clustering
  – Mapper outputs (cluster, docid) tuples; reducer adds up document representations to make centroid
• And many, many more