MapReduce and Hadoop

Debapriyo Majumdar
Indian Statistical Institute Kolkata
debapriyo@isical.ac.in
Let’s keep the intro short

- Modern data mining: process immense amount of data quickly
- Exploit parallelism

![Parallelism Diagram]

**Traditional parallelism**
Bring data to compute

**MapReduce**
Bring compute to data

Pictures courtesy: Glenn K. Lockwood, glennklockwood.com
The MapReduce paradigm

1. **Split**
   - original Input

2. **Map**
   - Input chunks
   - <Key, Value> pairs

3. **Shuffle and sort**
   - <Key, Value> pairs grouped by keys

4. **Reduce**
   - Output chunks

5. **Final**
   - Final output

- May be already split in filesystem
- The user needs to write the `map()` and the `reduce()`
- May not need to combine
An example: word frequency counting

Problem: Given a collection of documents, count the number of times each word occurs in the collection.

Input:
- Collection of documents
- Input chunks

Map:
- Map: for each word w, emit pairs (w,1)

Shuffle and sort:
- The pairs (w,1) for the same words are grouped together

Reduce:
- Reduce: count the number (n) of pairs for each w, make it (w,n)

Output:
- Output chunks
- Final output: (w,n) for each w
An example: word frequency counting

Problem: Given a collection of documents, count the number of times each word occurs in the collection.

Input chunks

map: for each word w, output pairs (w,1)

reduce: count the number (n) of pairs for each w, make it (w,n)

Output chunks

Final output

(apple,2) (orange,3) (plum,2) (cherry,2) (fig,2) (peach,3)
Apache Hadoop
An open source MapReduce framework

HADOOP
Hadoop

- Two main components
  - **Hadoop Distributed File System (HDFS)**: to store data
  - **MapReduce engine**: to process data
- Master – slave architecture using commodity servers

- **The HDFS**
  - Master: Namenode
  - Slave: Datanode
- **MapReduce**
  - Master: JobTracker
  - Slave: TaskTracker
HDFS: Blocks

- Runs on top of existing filesystem
- Blocks are 64MB (128MB recommended)
- Single file can be > any single disk
- POSIX based permissions
- Fault tolerant
HDFS: Namenode and Datanode

- **Namenode**
  - Only one per Hadoop Cluster
  - Manages the filesystem namespace
  - The filesystem tree
  - An edit log
  - For each block block $i$, the datanode(s) in which block $i$ is saved
  - All the blocks residing in each datanode

- **Secondary Namenode**
  - Backup namenode

- **Datanodes**
  - Many per Hadoop cluster
  - Controls block operations
  - Physically puts the block in the nodes
  - Do the physical replication
HDFS: an example
1. JobClient submits job to JobTracker; Binary copied into HDFS
2. JobTracker talks to Namenode
3. JobTracker creates execution plan
4. JobTracker submits work to TaskTrackers
5. TaskTrackers report progress via heartbeat
6. JobTracker updates status
Map, Shuffle and Reduce: internal steps

1. Splits data up to send it to the mapper
2. Transforms splits into key/value pairs
3. (Key-Value) with same key sent to the same reducer
4. Aggregates key/value pairs based on user-defined code
5. Determines how the result are saved
Fault Tolerance

- If the master fails
  - MapReduce would fail, have to restart the entire job
- A map worker node fails
  - Master detects (periodic ping would timeout)
  - All the map tasks for this node have to be restarted
    - Even if the map tasks were done, the output were at the node
- A reduce worker fails
  - Master sets the status of its currently executing reduce tasks to idle
  - Reschedule these tasks on another reduce worker
Some algorithms using MapReduce

USING MAPREDUCE
Matrix – Vector Multiplication

- Multiply $M = (m_{ij})$ (an $n \times n$ matrix) and $v = (v_j)$ (an $n$-vector)
- If $n = 1000$, no need of MapReduce!

If $n = 1000$, no need of MapReduce!

\[ Mv = (x_i) \]

\[ x_i = \sum_{j=1}^{n} m_{ij}v_j \]

Case 1: Large $n$, $M$ does not fit into main memory, but $v$ does

- Since $v$ fits into main memory, $v$ is available to every map task
- **Map**: for each matrix element $m_{ij}$, emit key value pair $(i, m_{ij}v_j)$
- **Shuffle and sort**: groups all $m_{ij}v_j$ values together for the same $i$
- **Reduce**: sum $m_{ij}v_j$ for all $j$ for the same $i$
Matrix – Vector Multiplication

- Multiply $M = (m_{ij})$ (an $n \times n$ matrix) and $v = (v_i)$ (an $n$-vector)
- If $n = 1000$, no need of MapReduce!

$$Mv = (x_i)$$

$$x_i = \sum_{j=1}^{n} m_{ij} v_j$$

Case 2: Very large $n$, even $v$ does not fit into main memory

- For every map, many accesses to disk (for parts of $v$) required!
- Solution:
  - **How much of $v$ will fit in?**
  - Partition $v$ and *rows* of $M$ so that each partition of $v$ fits into memory
  - Take dot product of one partition of $v$ and the corresponding partition of $M$
  - *Map* and *reduce* same as before
Relational Algebra

- Relation $R(A_1, A_3, \ldots, A_n)$ is a relation with attributes $A_i$
- Schema: set of attributes
- Selection on condition $C$: apply $C$ on each tuple in $R$, output only those which satisfy $C$
- Projection on a subset $S$ of attributes: output the components for the attributes in $S$
- Union, Intersection, Join…

<table>
<thead>
<tr>
<th>$Attr_1$</th>
<th>$Attr_2$</th>
<th>$Attr_3$</th>
<th>$Attr_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>xyz</td>
<td>abc</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>abc</td>
<td>xyz</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>xyz</td>
<td>def</td>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>bcd</td>
<td>def</td>
<td>2</td>
<td>true</td>
</tr>
</tbody>
</table>

Links between URLs

<table>
<thead>
<tr>
<th>$URL1$</th>
<th>$URL2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>url1</td>
<td>url2</td>
</tr>
<tr>
<td>url2</td>
<td>url1</td>
</tr>
<tr>
<td>url3</td>
<td>url5</td>
</tr>
<tr>
<td>url1</td>
<td>url3</td>
</tr>
</tbody>
</table>
Selection using MapReduce

- Trivial
- Map: For each tuple \( t \) in \( R \), test if \( t \) satisfies \( C \). If so, produce the key-value pair \((t, t)\).
- Reduce: The identity function. It simply passes each key-value pair to the output.

<table>
<thead>
<tr>
<th>URL1</th>
<th>URL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>url1</td>
<td>url2</td>
</tr>
<tr>
<td>url2</td>
<td>url1</td>
</tr>
<tr>
<td>url3</td>
<td>url5</td>
</tr>
<tr>
<td>url1</td>
<td>url3</td>
</tr>
</tbody>
</table>
Union using MapReduce

- Union of two relations $R$ and $S$
- Suppose $R$ and $S$ have the same schema
- Map tasks are generated from chunks of both $R$ and $S$
- Map: For each tuple $t$, produce the key-value pair $(t, t)$
- Reduce: Only need to remove duplicates
  - For all key $t$, there would be either one or two values
  - Output $(t, t)$ in either case

Links between URLs

<table>
<thead>
<tr>
<th>URL1</th>
<th>URL2</th>
</tr>
</thead>
<tbody>
<tr>
<td>url1</td>
<td>url2</td>
</tr>
<tr>
<td>url2</td>
<td>url1</td>
</tr>
<tr>
<td>url3</td>
<td>url5</td>
</tr>
<tr>
<td>url1</td>
<td>url3</td>
</tr>
</tbody>
</table>
Natural join using MapReduce

- Join R(A,B) with S(B,C) on attribute B
- Map:
  - For each tuple $t = (a,b)$ of $R$, emit key value pair $(b,(R,a))$
  - For each tuple $t = (b,c)$ of $S$, emit key value pair $(b,(S,c))$
- Reduce:
  - Each key $b$ would be associated with a list of values that are of the form $(R,a)$ or $(S,c)$
  - Construct all pairs consisting of one with first component $R$ and the other with first component $S$, say $(R,a)$ and $(S,c)$. The output from this key and value list is a sequence of key-value pairs
  - The key is irrelevant. Each value is one of the triples $(a, b, c)$ such that $(R,a)$ and $(S,c)$ are on the input list of values

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
Grouping and Aggregation using MapReduce

- Group and aggregate on a relation $R(A,B)$ using aggregation function $\gamma(B)$, group by $A$

- Map:
  - For each tuple $t = (a,b)$ of $R$, emit key value pair $(a,b)$

- Reduce:
  - For all group $\{(a,b_1), \ldots, (a,b_m)\}$ represented by a key $a$, apply $\gamma$ to obtain $b_a = b_1 + \ldots + b_m$
  - Output $(a,b_a)$

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>2</td>
</tr>
<tr>
<td>$y$</td>
<td>1</td>
</tr>
<tr>
<td>$z$</td>
<td>4</td>
</tr>
<tr>
<td>$z$</td>
<td>1</td>
</tr>
<tr>
<td>$x$</td>
<td>5</td>
</tr>
</tbody>
</table>

$R$ select $A$, $\text{sum}(B)$ from $R$ group by $A$;

<table>
<thead>
<tr>
<th>$A$</th>
<th>$\text{SUM}(B)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>7</td>
</tr>
<tr>
<td>$y$</td>
<td>1</td>
</tr>
<tr>
<td>$z$</td>
<td>5</td>
</tr>
</tbody>
</table>
Matrix multiplication using MapReduce

Think of a matrix as a relation with three attributes

- For example matrix $A$ is represented by the relation $A(I, J, V)$
  - For every non-zero entry $(i, j, a_{ij})$, the row number is the value of $I$, column number is the value of $J$, the entry is the value in $V$
  - Also advantage: usually most large matrices would be sparse, the relation would have less number of entries

The product is $\sim$ a natural join followed by a grouping with aggregation

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}$$
Matrix multiplication using MapReduce

Matrix multiplication:

\[ A \times B = C \]

where:

- \( A \) is an \( m \times n \) matrix
- \( B \) is an \( n \times l \) matrix
- \( C \) is an \( m \times l \) matrix

\[ c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk} \]

Natural join of \((I,J,V)\) and \((J,K,W)\) \(\rightarrow\) tuples \((i, j, k, a_{ij}, b_{jk})\)

Map:

- For every \((i, j, a_{ij})\), emit key value pair \((j, (A, i, a_{ij}))\)
- For every \((j, k, b_{jk})\), emit key value pair \((j, (B, k, b_{jk}))\)

Reduce:

for each key \(j\)

for each value \((A, i, a_{ij})\) and \((B, k, b_{jk})\)

produce a key value pair \(((i,k),(a_{ij}b_{jk}))\)
Matrix multiplication using MapReduce

**Matrix**

\[
A = \begin{pmatrix}
m \\ (m \times n) \\ (i, j, a_{ij})
\end{pmatrix}
\]

\[
B = \begin{pmatrix}
l \\ (n \times l) \\ (j, k, b_{jk})
\end{pmatrix}
\]

\[
C = \begin{pmatrix}
l \\ (m \times l)
\end{pmatrix}
\]

\[c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}\]

- First MapReduce process has produced key value pairs \(((i,k), (a_{ij}b_{jk}))\)
- Another MapReduce process to group and aggregate
- Map: identity, just emit the key value pair \(((i,k), (a_{ij}b_{jk}))\)
- Reduce:
  - for each key \((i,k)\)
  - produce the sum of the all the values for the key: \[c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}\]
Matrix multiplication using MapReduce: Method 2

- A method with one MapReduce step
- Map:
  - For every \((i, j, a_{ij})\), emit for all \(k = 1, \ldots, l\), the key value \(((i,k), (A, j, a_{ij}))\)
  - For every \((j, k, b_{jk})\), emit for all \(i = 1, \ldots, m\), the key value \(((i,k), (B, j, b_{jk}))\)
- Reduce:
  
  for each key \((i,k)\)
  sort values \((A, j, a_{ij})\) and \((B, j, b_{jk})\) by \(j\) to group them by \(j\)
  for each \(j\) multiply \(a_{ij}\) and \(b_{jk}\)
  sum the products for the key \((i,k)\) to produce \(c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}\)

May not fit in main memory. Expensive external sort!
Combiners

- If the aggregation operator in the reduce algorithm is associative and commutative
  - Associative: \((a \ast b) \ast c = a \ast (b \ast c)\)
  - Commutative: \(a \ast b = b \ast a\)
  - It does not matter in which order we aggregate

- We can do some work for reduce in the map step

- Example: the word count problem
  - Instead of emitting \((w, 1)\) for each word
  - We can emit \((w, n)\) for each document or each map task
More examples
An inverted index for text collections
### Collection and Documents

<table>
<thead>
<tr>
<th>Document</th>
<th>Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>The curse of the black pearl</em></td>
<td>Ship Jack Sparrow, Caribbean Turner, Elizabeth Gun Fight</td>
</tr>
<tr>
<td><em>Finding Nemo</em></td>
<td>Ocean Fish Nemo, Reef Animation</td>
</tr>
<tr>
<td><em>Tintin</em></td>
<td>Ocean Animation, Ship Haddock, Tintin</td>
</tr>
<tr>
<td><em>Titanic</em></td>
<td>Ship Rose Jack, Atlantic Ocean, England Sink</td>
</tr>
<tr>
<td><em>The Dark Knight</em></td>
<td>Bruce Wayne Batman, Joker Harvey Gordon, Gun Fight Crime</td>
</tr>
<tr>
<td><em>Skyfall</em></td>
<td>007 James Bond, MI6 Gun Fight</td>
</tr>
<tr>
<td><em>Silence of the Lambs</em></td>
<td>Hannibal Lector, FBI Crime, Gun Cannibal</td>
</tr>
<tr>
<td><em>The Ghost Ship</em></td>
<td>Ship Ghost Ocean, Death Horror</td>
</tr>
</tbody>
</table>

- Document: unit of retrieval
- Collection: the group of documents from which we retrieve
  - Also called *corpus* (a body of texts)
Boolean retrieval

- Find all documents containing a word $w$
- Find all documents containing a word $w_1$ but not containing the word $w_2$
- Queries in the form of any Boolean expression
- Query: Jack
Boolean retrieval

- Find all documents containing a word $w$
- Find all documents containing a word $w_1$ but not containing the word $w_2$
- Queries in the form of any Boolean expression
- Query: Jack
Inverted index

1. Black pearl
2. Finding Nemo
3. Tintin
4. Titanic
5. Dark Knight
6. Skyfall
7. Silence of lambs
8. Ghost ship

Ship
Jack
Bond
Gun
Ocean
Captain
Batman
Crime
Inverted index

Inverted Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Posting lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ship</td>
<td>1 3 4 8</td>
</tr>
<tr>
<td>Jack</td>
<td>1 4</td>
</tr>
<tr>
<td>Bond</td>
<td>6</td>
</tr>
<tr>
<td>Gun</td>
<td>1 5 6 7</td>
</tr>
<tr>
<td>Ocean</td>
<td>1 2 3 4 8</td>
</tr>
<tr>
<td>Captain</td>
<td>1 3 4</td>
</tr>
<tr>
<td>Batman</td>
<td>5</td>
</tr>
<tr>
<td>Crime</td>
<td>5 7</td>
</tr>
</tbody>
</table>
Creating an inverted index

Map: for each term in each document, write out pairs (term, docid)

Reduce: List documents for each term
A problem: find people you may know

In a social network

- There are several factors
- A very important one is by number of mutual friends
- How to compute a list of suggestions?
  - Is one mutual friend enough?
  - If yes, then simply all friends of friends
  - Sometimes, but more is better
- Consider a simple criterion:

If X and Y have at least 3 mutual friends then Y is a suggested friend of X
The Data

- For every member, there is a list of friends

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>→</td>
<td>B</td>
<td>C</td>
<td>L</td>
</tr>
<tr>
<td>B</td>
<td>→</td>
<td>A</td>
<td>G</td>
<td>K</td>
</tr>
<tr>
<td>C</td>
<td>→</td>
<td>A</td>
<td>G</td>
<td>M</td>
</tr>
</tbody>
</table>

Naïve approach:
- For every friend X of A
  - For every friend Y of X
    - Compute number of mutual friends between Y and A
    - Intersect the friend-list of Y and friend-list of A

\(O(n^3)\) per member. Total \(O(N n^3)!!\)

\(N\) members.
Every member has \(n\) friends. \(N >> n\)
### A better method

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>L</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>A</td>
<td>G</td>
<td>K</td>
<td>...</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>G</td>
<td>M</td>
<td>...</td>
</tr>
</tbody>
</table>

- For each friend X of A:
  - For each friend Y of X:
    - Write: (A, Y, X)

- What do we get?
- Now sort by first two entries
  - All the (A,G)’s are together
- For each (A, X), count how many times it occurs:
  - Number of mutual friends between A and X

O(\(n^2\log n\)) per member
Challenges

- For each friend X of A
  - For each friend Y of X
    - Write: (A, Y, X)
- Now sort by first two entries
  - All the (A,G)’s are together
- For each (A, Y), count how many times it occurs

Data may be too huge, distributed
Friendlist of A in one part of storage
Friendlist of friend of A in another part
How to do this using MapReduce?

- Assume the friendlist for each user is in one machine.
- Describe an algorithm using MapReduce.
Extensions of MapReduce

- MapReduce: two-step workflow
- Extension: any collection of functions
  - Acyclic graph representing the workflow

- Each function of the workflow can be executed by many tasks
- A master controller divides the work among the tasks
- Each task’s output goes to the successor as input
- Two experimental systems: Clustera (University of Wisconsin), Hyracks (University of California at Irvine)
- Similarly, recursive use of MapReduce
Cost of MapReduce based algorithms

- Computation in the nodes → Computation cost
- Transferring of data through network → Communication cost
- Communication cost is most often the bottleneck. Why?
Why communication cost?

- The algorithm executed by each task is usually very simple, often linear in the size of its input
- Network speed within cluster ~ 1Gigabit/s
  - But processor executes simple map or reduce tasks even faster
  - In a cluster several network transfers may be required at the same time → overload
- Data typically stored in disk
  - Reading the data into main memory may take more time than to process it in map or reduce tasks
Communication cost

- Communication cost of a task = The size of the input to the task

- Why size of the input?
  - Most often size of the input ≥ size of the output
  - Why?
    - Because unless summarized, the output can’t be used much
    - If output size is large, then it is the input for another task (counting input size suffices)
References and acknowledgements

- Mining of Massive Datasets, by Leskovec, Rajaraman and Ullman, Chapter 2
- Slides by Dwaipayan Roy