Data-Intensive Information Processing Applications — Session #3

MapReduce Algorithm Design

Jimmy Lin
University of Maryland

Tuesday, February 9, 2010
Today’s Agenda

- “The datacenter is the computer”
  - Understanding the design of warehouse-sized computers

- MapReduce algorithm design
  - How do you express everything in terms of m, r, c, p?
  - Toward “design patterns”
The datacenter *is* the computer
“Big Ideas”

- Scale “out”, not “up”
  - Limits of SMP and large shared-memory machines
- Move processing to the data
  - Cluster have limited bandwidth
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable
- Seamless scalability
  - From the mythical man-month to the tradable machine-hour
Building Blocks

Source: Barroso and Urs Hölzle (2009)
Storage Hierarchy

Funny story about sense of scale…

Source: Barroso and Urs Hölzle (2009)
Storage Hierarchy

Source: Barroso and Urs Hölzle (2009)
Anatomy of a Datacenter

Power Distribution Unit (PDU)
- Typical Capacities Up To 225 kVA Per Unit
- Redundancy Through Dual PDU’s With Integral Static Transfer Switch (STS)

Computer Air Handling Unit (CRAC)
- Up To 30 Ton Sensible Capacity Per Unit
- Air Discharge Can Be Upflow Or Downflow Configuration
- Downflow Configuration Used With Raised Floor To Create A Pressurized Supply Air Plenum With Floor Supply Diffusers

Individual Colocation Computer Cabinets
- Typ. Cabinet Footprint (28”W x 36”D x 84”H)
- Typical Capacities Of 1750 To 3750 Watts Per Cabinet

Colocation Suites
- Modular Configuration For Flexible Suite Sq Ft Areas.
- Suites Consist Of Multiple Cabinets With Secured Partitions (Cages, Walls, Etc.)

Emergency Diesel Generators
- Total Generator Capacity = Total Electrical Load To Building
- Multiple Generators Can Be Electrically Combined With Parallel Gear
- Can Be Located Indoors Or Outdoors At Grade Or On Roof
- Outdoor Applications Require Sound Attenuating Enclosures

Fuel Oil Storage Tanks
- Tank Capacity Dependent On Length Of Generator Operation
- Can Be Located Underground Or At Grade Or Indoors

UPS System
- Uninterruptible Power Supply Modules
- Up To 1000 kVA Per Module
- Cabinets And Battery Strings Or Rotary Flywheels
- Multiple Redundancy Configurations Can Be Designed

Electrical Primary Switchgear
- Includes Incoming Service And Distribution
- Direct Distribution To Mechanical Equipment
- Distribution To Secondary Electrical Equipment Via UPS

Heat Rejection Devices
- Drycoolers, Air Cooled Chillers, Etc.
- Up To 400 Ton Capacity Per Unit
- Mounted At Grade Or On Roof
- N+1 Design

Pump Room
- Used To Pump Condenser/Chilled Water Between Drycoolers And CRAC Units
- Additional Equipment Includes Expansion Tank, Glycol Feed System
- N+1 Design (Standby Pump)

Source: Barroso and Urs Hölzle (2009)
# Why commodity machines?

<table>
<thead>
<tr>
<th></th>
<th>HP INTEGRITY SUPERDOME-ITANIUM2</th>
<th>HP PROLIANT ML350 G5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor</strong></td>
<td>64 sockets, 128 cores (dual-threaded), 1.6 GHz Itanium2, 12 MB last-level cache</td>
<td>1 socket, quad-core, 2.66 GHz X5355 CPU, 8 MB last-level cache</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>2,048 GB</td>
<td>24 GB</td>
</tr>
<tr>
<td><strong>Disk storage</strong></td>
<td>320,974 GB, 7,056 drives</td>
<td>3,961 GB, 105 drives</td>
</tr>
<tr>
<td><strong>TPC-C price/performance</strong></td>
<td>$2.93/tpmC</td>
<td>$0.73/tpmC</td>
</tr>
<tr>
<td><strong>price/performance</strong></td>
<td>$1.28/transactions per minute</td>
<td>$0.10/transactions per minute</td>
</tr>
<tr>
<td><strong>(server HW only)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Price/performance</strong></td>
<td>$2.39/transactions per minute</td>
<td>$0.12/transactions per minute</td>
</tr>
<tr>
<td><strong>(server HW only, no discounts)</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: Barroso and Urs Hölzle (2009); performance figures from late 2007
What about communication?

- Nodes need to talk to each other!
  - SMP: latencies \(\sim 100\) ns
  - LAN: latencies \(\sim 100\) \(\mu\)s

- Scaling “up” vs. scaling “out”
  - Smaller cluster of SMP machines vs. larger cluster of commodity machines
  - E.g., 8 128-core machines vs. 128 8-core machines
  - Note: no single SMP machine is big enough

- Let’s model communication overhead…

Source: analysis on this an subsequent slides from Barroso and Urs Hölzle (2009)
Modeling Communication Costs

- Simple execution cost model:
  - Total cost = cost of computation + cost to access global data
  - Fraction of local access inversely proportional to size of cluster
  - $n$ nodes (ignore cores for now)

  $$1 \text{ ms} + f \times [100 \text{ ns} \times n + 100 \text{ } \mu\text{s} \times (1 - 1/n)]$$

  - Light communication: $f = 1$
  - Medium communication: $f = 10$
  - Heavy communication: $f = 100$

- What are the costs in parallelization?
Cost of Parallelization

![Graph showing the cost of parallelization with different communication levels: high, medium, and light. The graph plots normalized execution time against the number of nodes.](Image)
Advantages of scaling “up”

So why not?
Seeks vs. Scans

- Consider a 1 TB database with 100 byte records
  - We want to update 1 percent of the records

- Scenario 1: random access
  - Each update takes ~30 ms (seek, read, write)
  - $10^8$ updates = ~35 days

- Scenario 2: rewrite all records
  - Assume 100 MB/s throughput
  - Time = 5.6 hours(!)

- Lesson: avoid random seeks!

Source: Ted Dunning, on Hadoop mailing list
Justifying the “Big Ideas”

- Scale “out”, not “up”
  - Limits of SMP and large shared-memory machines
- Move processing to the data
  - Cluster have limited bandwidth
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable
- Seamless scalability
  - From the mythical man-month to the tradable machine-hour
Numbers Everyone Should Know*

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>20,000,000</td>
</tr>
<tr>
<td>Send packet CA → Netherlands → CA</td>
<td>150,000,000</td>
</tr>
</tbody>
</table>

* According to Jeff Dean (LADIS 2009 keynote)
MapReduce Algorithm Design
**MapReduce: Recap**

- Programmers must specify:
  
  map \((k, v) \rightarrow <k', v'>\)*
  
  reduce \((k', v') \rightarrow <k', v'>\)*
  
  - All values with the same key are reduced together

- Optionally, also:
  
  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

- The execution framework handles everything else…
Shuffle and Sort: aggregate values by keys
“Everything Else”

- The execution framework handles everything else…
  - Scheduling: assigns workers to map and reduce tasks
  - “Data distribution”: moves processes to data
  - Synchronization: gathers, sorts, and shuffles intermediate data
  - Errors and faults: detects worker failures and restarts

- Limited control over data and execution flow
  - All algorithms must expressed in m, r, c, p

- You don’t know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing
Tools for Synchronization

- Cleverly-constructed data structures
  - Bring partial results together
- Sort order of intermediate keys
  - Control order in which reducers process keys
- Partitioner
  - Control which reducer processes which keys
- Preserving state in mappers and reducers
  - Capture dependencies across multiple keys and values
Preserving State

Mapper object

- state
- configure
- map
- close

Reducer object

- state
- configure
- reduce
- close

- API initialization hook
- one call per input key-value pair
- one call per intermediate key
- API cleanup hook

- one object per task
Scalable Hadoop Algorithms: Themes

- Avoid object creation
  - Inherently costly operation
  - Garbage collection

- Avoid buffering
  - Limited heap size
  - Works for small datasets, but won’t scale!
Importance of Local Aggregation

- Ideal scaling characteristics:
  - Twice the data, twice the running time
  - Twice the resources, half the running time

- Why can’t we achieve this?
  - Synchronization requires communication
  - Communication kills performance

- Thus… avoid communication!
  - Reduce intermediate data via local aggregation
  - Combiners can help
Shuffle and Sort

Mapper

- circular buffer (in memory)
- spills (on disk)

Reducer

- intermediate files (on disk)

Combiner

- merged spills (on disk)

Other mappers

- spills (on disk)

Other reducers

- spills (on disk)
What’s the impact of combiners?
Word Count: Version 1

1: class MAPPER
2: method MAP(docid a, doc d)
3:     H ← new ASSOCIATIVEARRAY
4:     for all term t ∈ doc d do
5:         H{t} ← H{t} + 1  \hspace{1cm} ▷ Tally counts for entire document
6:     for all term t ∈ H do
7:         EMIT(term t, count H{t})

Are combiners still needed?
class Mapper

method INITIALIZ

\[ H \leftarrow \text{new ASSOCIATIVEARRAY} \]

method MAP(docid \( a \), doc \( d \))

\begin{align*}
\text{for all term } t \in \text{doc } d & \text{ do} \\
H\{t\} & \leftarrow H\{t\} + 1
\end{align*}

\[ \text{▷ Tally counts across documents} \]

method CLOSE

\begin{align*}
\text{for all term } t \in H & \text{ do} \\
\text{EMIT(term } t, \text{ count } H\{t\})
\end{align*}

Are combiners still needed?
Design Pattern for Local Aggregation

○ “In-mapper combining”
  ● Fold the functionality of the combiner into the mapper by preserving state across multiple map calls

○ Advantages
  ● Speed
  ● Why is this faster than actual combiners?

○ Disadvantages
  ● Explicit memory management required
  ● Potential for order-dependent bugs
Combiner Design

- Combiners and reducers share same method signature
  - Sometimes, reducers can serve as combiners
  - Often, not…

- Remember: combiner are optional optimizations
  - Should not affect algorithm correctness
  - May be run 0, 1, or multiple times

- Example: find average of all integers associated with the same key
Computing the Mean: Version 1

1: class Mapper
2:   method Map(string t, integer r)
3:     Emit(string t, integer r)

1: class Reducer
2:   method Reduce(string t, integers [r₁, r₂, ...])
3:     sum ← 0
4:     cnt ← 0
5:     for all integer r ∈ integers [r₁, r₂, ...] do
6:       sum ← sum + r
7:       cnt ← cnt + 1
8:     r_avg ← sum/cnt
9:     Emit(string t, integer r_avg)

Why can’t we use reducer as combiner?
Computing the Mean: Version 2

1: class Mapper
2:   method Map(string t, integer r)
3:     Emit(string t, integer r)

1: class Combiner
2:   method Combine(string t, integers [r₁, r₂, ...])
3:     sum ← 0
4:     cnt ← 0
5:     for all integer r ∈ integers [r₁, r₂, ...] do
6:       sum ← sum + r
7:       cnt ← cnt + 1
8:     Emit(string t, pair (sum, cnt))  ▷ Separate sum and count

1: class Reducer
2:   method Reduce(string t, pairs [(s₁, c₁), (s₂, c₂) ...])
3:     sum ← 0
4:     cnt ← 0
5:     for all pair (s, c) ∈ pairs [(s₁, c₁), (s₂, c₂) ...] do
6:       sum ← sum + s
7:       cnt ← cnt + c
8:       ravg ← sum/cnt
9:     Emit(string t, integer ravg)

Why doesn’t this work?
Computing the Mean: Version 3

```java
1: class MAPPER
2:     method Map(string t, integer r)
3:         Emit(string t, pair (r, 1))

1: class COMBINER
2:     method Combine(string t, pairs [(s1, c1), (s2, c2), ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2), ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         Emit(string t, pair (sum, cnt))

1: class REDUCER
2:     method Reduce(string t, pairs [(s1, c1), (s2, c2), ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2), ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_{avg} ← sum/cnt
9:         Emit(string t, pair (r_{avg}, cnt))
```

Fixed?
Computing the Mean: Version 4

1: class Mapper
2:     method Initialize
3:         S ← new AssociativeArray
4:         C ← new AssociativeArray
5:     method Map(string t, integer r)
6:         S{t} ← S{t} + r
7:         C{t} ← C{t} + 1
8:     method Close
9:         for all term t ∈ S do
10:             Emit(term t, pair (S{t}, C{t}))

Are combiners still needed?
Algorithm Design: Running Example

- Term co-occurrence matrix for a text collection
  - $M = N \times N$ matrix ($N =$ vocabulary size)
  - $M_{ij}$: number of times $i$ and $j$ co-occur in some context
    (for concreteness, let’s say context = sentence)

- Why?
  - Distributional profiles as a way of measuring semantic distance
  - Semantic distance useful for many language processing tasks
MapReduce: Large Counting Problems

- Term co-occurrence matrix for a text collection
  = specific instance of a large counting problem
  ● A large event space (number of terms)
  ● A large number of observations (the collection itself)
  ● Goal: keep track of interesting statistics about the events

- Basic approach
  ● Mappers generate partial counts
  ● Reducers aggregate partial counts

How do we aggregate partial counts efficiently?
First Try: “Pairs”

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For all pairs, emit \((a, b) \rightarrow \text{count}\)

- Reducers sum up counts associated with these pairs

- Use combiners!
Pairs: Pseudo-Code

1: class Mapper
2:  method MAP(docid a, doc d)
3:      for all term w ∈ doc d do
4:         for all term u ∈ NEIGHBORS(w) do
5:            EMIT(pair (w, u), count 1)  \> Emit count for each co-occurrence

1: class Reducer
2:  method Reduce(pair p, counts [c₁, c₂, …])
3:      s ← 0
4:      for all count c ∈ counts [c₁, c₂, …] do
5:         s ← s + c  \> Sum co-occurrence counts
6:      EMIT(pair p, count s)
“Pairs” Analysis

- Advantages
  - Easy to implement, easy to understand

- Disadvantages
  - Lots of pairs to sort and shuffle around (upper bound?)
  - Not many opportunities for combiners to work
Another Try: “Stripes”

- Idea: group together pairs into an associative array
  
  \[
  \begin{align*}
  (a, b) & \to 1 \\
  (a, c) & \to 2 \\
  (a, d) & \to 5 \\
  (a, e) & \to 3 \\
  (a, f) & \to 2 
  \end{align*}
  \]

- Each mapper takes a sentence:
  
  - Generate all co-occurring term pairs
  - For each term, emit \( a \to \{ b: \text{count}_b, c: \text{count}_c, d: \text{count}_d \ldots \} \)

-Reducers perform element-wise sum of associative arrays

  \[
  \begin{align*}
  a \to \{ b: 1, & \quad d: 5, e: 3 \} \\
  + a \to \{ b: 1, & \quad c: 2, d: 2, f: 2 \} \\
  \hline
  a \to \{ b: 2, & \quad c: 2, d: 7, e: 3, f: 2 \}
  \end{align*}
  \]

Key: cleverly-constructed data structure brings together partial results
Stripes: Pseudo-Code

1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             H ← new AssociativeArray
5:                 for all term u ∈ Neighbors(w) do
6:                     H{u} ← H{u} + 1 \quad \triangleright \text{Tally words co-occurring with } w
7:                 Emit(Term w, Stripe H)

1: class Reducer
2:     method Reduce(term w, stripes [H₁, H₂, H₃, ...])
3:         H_f ← new AssociativeArray
4:             for all stripe H ∈ stripes [H₁, H₂, H₃, ...] do
5:                 Sum(H_f, H) \quad \triangleright \text{Element-wise sum}
6:             Emit(term w, stripe H_f)
“Stripes” Analysis

- Advantages
  - Far less sorting and shuffling of key-value pairs
  - Can make better use of combiners

- Disadvantages
  - More difficult to implement
  - Underlying object more heavyweight
  - Fundamental limitation in terms of size of event space
Comparison of "pairs" vs. "stripes" for computing word co-occurrence matrices

Cluster size: 38 cores
Data Source: Associated Press Worldstream (APW) of the English Gigaword Corpus (v3), which contains 2.27 million documents (1.8 GB compressed, 5.7 GB uncompressed)
Effect of cluster size on "stripes" algorithm

relative size of EC2 cluster

running time (seconds)

relative speedup

size of EC2 cluster (number of slave instances)

$R^2 = 0.997$
Relative Frequencies

- How do we estimate relative frequencies from counts?

\[ f(B \mid A) = \frac{\text{count}(A, B)}{\text{count}(A)} = \frac{\text{count}(A, B)}{\sum_{B'} \text{count}(A, B')} \]

- Why do we want to do this?

- How do we do this with MapReduce?
f(B|A): “Stripes”

\[ a \rightarrow \{ b_1 : 3, b_2 : 12, b_3 : 7, b_4 : 1, \ldots \} \]

- Easy!
  - One pass to compute (a, *)
  - Another pass to directly compute f(B|A)
### f(B|A): “Pairs”

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a, *)</td>
<td>32</td>
</tr>
<tr>
<td>(a, b₁)</td>
<td>3</td>
</tr>
<tr>
<td>(a, b₂)</td>
<td>12</td>
</tr>
<tr>
<td>(a, b₃)</td>
<td>7</td>
</tr>
<tr>
<td>(a, b₄)</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Reducer holds this value in memory

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a, b₁)</td>
<td>3 / 32</td>
</tr>
<tr>
<td>(a, b₂)</td>
<td>12 / 32</td>
</tr>
<tr>
<td>(a, b₃)</td>
<td>7 / 32</td>
</tr>
<tr>
<td>(a, b₄)</td>
<td>1 / 32</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- For this to work:
  - Must emit extra (a, *) for every bᵣ in mapper
  - Must make sure all a’s get sent to same reducer (use partitioner)
  - Must make sure (a, *) comes first (define sort order)
  - Must hold state in reducer across different key-value pairs
“Order Inversion”

- Common design pattern
  - Computing relative frequencies requires marginal counts
  - But marginal cannot be computed until you see all counts
  - Buffering is a bad idea!
  - Trick: getting the marginal counts to arrive at the reducer before the joint counts

- Optimizations
  - Apply in-memory combining pattern to accumulate marginal counts
  - Should we apply combiners?
Synchronization: Pairs vs. Stripes

- Approach 1: turn synchronization into an ordering problem
  - Sort keys into correct order of computation
  - Partition key space so that each reducer gets the appropriate set of partial results
  - Hold state in reducer across multiple key-value pairs to perform computation
  - Illustrated by the “pairs” approach

- Approach 2: construct data structures that bring partial results together
  - Each reducer receives all the data it needs to complete the computation
  - Illustrated by the “stripes” approach
Secondary Sorting

- MapReduce sorts input to reducers by key
  - Values may be arbitrarily ordered
- What if want to sort value also?
  - E.g., \(k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r)\)...
Secondary Sorting: Solutions

- Solution 1:
  - Buffer values in memory, then sort
  - Why is this a bad idea?

- Solution 2:
  - “Value-to-key conversion” design pattern: form composite intermediate key, \((k, v_1)\)
  - Let execution framework do the sorting
  - Preserve state across multiple key-value pairs to handle processing
  - Anything else we need to do?
Recap: Tools for Synchronization

- Cleverly-constructed data structures
  - Bring data together
- Sort order of intermediate keys
  - Control order in which reducers process keys
- Partitioner
  - Control which reducer processes which keys
- Preserving state in mappers and reducers
  - Capture dependencies across multiple keys and values
Issues and Tradeoffs

- Number of key-value pairs
  - Object creation overhead
  - Time for sorting and shuffling pairs across the network

- Size of each key-value pair
  - De/serialization overhead

- Local aggregation
  - Opportunities to perform local aggregation varies
  - Combiners make a big difference
  - Combiners vs. in-mapper combining
  - RAM vs. disk vs. network
Debugging at Scale

- Works on small datasets, won’t scale… why?
  - Memory management issues (buffering and object creation)
  - Too much intermediate data
  - Mangled input records
- Real-world data is messy!
  - Word count: how many unique words in Wikipedia?
  - There’s no such thing as “consistent data”
  - Watch out for corner cases
  - Isolate unexpected behavior, bring local
Questions?