Big Data Systems
Big Data Parallelism

- Huge data set
  - crawled documents, web request logs, etc.
- Natural parallelism:
  - can work on different parts of data independently
  - image processing, grep, indexing, many more
Challenges

• Parallelize application
  • Where to place input and output data?
  • Where to place computation?
  • How to communicate data? How to manage threads? How to avoid network bottlenecks?

• Balance computations

• Handle failures of nodes during computation

• Scheduling several applications who want to share infrastructure
Goal of MapReduce

- To solve these distribution/fault-tolerance issues once in a reusable library
- To shield the programmer from having to re-solve them for each program
- To obtain adequate throughput and scalability
- To provide the programmer with a conceptual framework for designing their parallel program
Map Reduce

Overview:
- Partition large data set into M splits
- Run map on each partition, which produces R local partitions; using a partition function R
- Hidden intermediate shuffle phase
- Run reduce on each intermediate partition, which produces R output files
Details

- Input values: set of key-value pairs
  - Job will read chunks of key-value pairs
  - “key-value” pairs a good enough abstraction

- Map(key, value):
  - System will execute this function on each key-value pair
  - Generate a set of intermediate key-value pairs

- Reduce(key, values):
  - Intermediate key-value pairs are sorted
  - Reduce function is executed on these intermediate key-values
Count words in web-pages

Map(key, value) {
    // key is url
    // value is the content of the url
    For each word W in the content
        Generate(W, 1);
}

Reduce(key, values) {
    // key is word (W)
    // values are basically all 1s
    Sum = Sum all 1s in values

    // generate word-count pairs
    Generate (key, sum);
}
Reverse web-link graph

Go to google advanced search: "find pages that link to the page:" cnn.com

Map(key, value) {
    // key = url
    // value = content
    For each url, linking to target
        Generate(output target, url);
}

Reduce(key, values) {
    // key = target url
    // values = all urls that point to the target url
    Generate(key, list of values);
}

Question: how do we implement “join” in MapReduce?

- Imagine you have a log table L and some other table R that contains say user information
- Perform Join (L.uid == R.uid)
  - Say size of L >> size of R
- Bonus: consider real world zipf distributions
Comparisons

- Worth comparing it to other programming models:
  - distributed shared memory systems
  - bulk synchronous parallel programs
  - key-value storage accessed by general programs
- More constrained programming model for MapReduce
- Other models are latency sensitive, have poor throughput efficiency
- MapReduce provides for easy fault recovery
Implementation

- Depends on the underlying hardware: shared memory, message passing, NUMA shared memory, etc.

- Inside Google:
  - commodity workstations
  - commodity networking hardware (1Gbps - 10Gbps now - at node level and much smaller bisection bandwidth)
  - cluster = 100s or 1000s of machines
  - storage is through GFS
MapReduce Input

- Where does input come from?
  - Input is striped+replicated over GFS in 64 MB chunks
  - But in fact Map always reads from a local disk
  - They run the Maps on the GFS server that holds the data

- Tradeoff:
  - Good: Map reads at disk speed (local access)
  - Bad: only two or three choices of where a given Map can run
    - potential problem for load balance, stragglers
Intermediate Data

- Where does MapReduce store intermediate data?
  - On the local disk of the Map server (not in GFS)

- Tradeoff:
  - Good: local disk write is faster than writing over network to GFS server
  - Bad: only one copy, potential problem for fault-tolerance and load-balance
Output Storage

- Where does MapReduce store output?
  - In GFS, replicated, separate file per Reduce task
  - So output requires network communication -- slow
  - It can then be used as input for subsequent MapReduce
Question

- What are the scalability bottlenecks for MapReduce?
Scaling

- Map calls probably scale
  - but input might not be infinitely partitionable, and small input/intermediate files incur high overheads
- Reduce calls probably scale
  - but can’t have more workers than keys, and some keys could have more values than others
- Network may limit scaling
- Stragglers could be a problem
Fault Tolerance

- The main idea: Map and Reduce are deterministic, functional, and independent
  - so MapReduce can deal with failures by re-executing
- What if a worker fails while running Map?
  - Can we restart just that Map on another machine?
  - Yes: GFS keeps copy of each input split on 3 machines
  - Master knows, tells Reduce workers where to find intermediate files
Fault Tolerance

- If a Map finishes, then that worker fails, do we need to re-run that Map?
  - Intermediate output now inaccessible on worker's local disk.
  - Thus need to re-run Map elsewhere *unless* all Reduce workers have already fetched that Map's output.

- What if Map had started to produce output, then crashed?
  - Need to ensure that Reduce does not consume the output twice

- What if a worker fails while running Reduce?
Role of the Master

- Keeps state regarding the state of each worker machine (pings each machine)
- Reschedules work corresponding to failed machines
- Orchestrates the passing of locations to reduce functions
Load Balance

- What if some Map machines are faster than others?
  - Or some input splits take longer to process?
  - Solution: many more input splits than machines
  - Master hands out more Map tasks as machines finish
  - Thus faster machines do bigger share of work

- But there's a constraint:
  - Want to run Map task on machine that stores input data
  - GFS keeps 3 replicas of each input data split
  - only three efficient choices of where to run each Map task
Stragglers

- Often one machine is slow at finishing very last task
  - bad hardware, overloaded with some other work
- Load balance only balances newly assigned tasks
- Solution: always schedule multiple copies of very last tasks!
How many MR tasks?

- Paper uses $M = 10x$ number of workers, $R = 2x$.
  - More =>
    - finer grained load balance.
    - less redundant work for straggler reduction.
    - spread tasks of failed worker over more machines
    - overlap Map and shuffle, shuffle and Reduce.
- Less => big intermediate files w/ less overhead.
- $M$ and $R$ also maybe constrained by how data is striped in GFS (e.g., 64MB chunks)
Discussion

- what are the constraints imposed on map and reduce functions?
- how would you like to expand the capability of map reduce?
Map Reduce Criticism

- “Giant step backwards” in programming model
- Sub-optimal implementation
- “Not novel at all”
- Missing most of the DB features
- Incompatible with all of the DB tools
Comparison to Databases

- Huge source of controversy; claims:
  - parallel databases have much more advanced data processing support that leads to much more efficiency
  - support an index; selection is accelerated
  - provides query optimization
  - parallel databases support a much richer semantic model
  - support a schema; sharing across apps
  - support SQL, efficient joins, etc.
Where does MR win?

- Scaling
- Loading data into system
- Fault tolerance (partial restarts)
- Approachability
Spark Motivation

- MR Problems
  - cannot support complex applications **efficiently**
  - cannot support interactive applications **efficiently**

- Root cause
  - Inefficient data sharing

In MapReduce, the only way to share data across jobs is stable storage -> **slow!**
Motivation

Slow due to replication and disk I/O, but necessary for fault tolerance
Goal: In-Memory Data Sharing
Challenge

- How to design a distributed memory abstraction that is both fault tolerant and efficient?
Other options

- Existing storage abstractions have interfaces based on fine-grained updates to mutable state
  - E.g., RAMCloud, databases, distributed mem, Piccolo
- Requires replicating data or logs across nodes for fault tolerance
  - Costly for data-intensive apps
  - 10-100x slower than memory write
RDD Abstraction

- Restricted form of distributed shared memory
  - immutable, partitioned collection of records
  - can only be built through coarse-grained deterministic transformations (map, filter, join...)

- Efficient fault-tolerance using lineage
  - Log coarse-grained operations instead of fine-grained data updates
  - An RDD has enough information about how it’s derived from other dataset
  - Recompute lost partitions on failure
Fault-tolerance
Design Space
Operations

- Transformations (e.g. map, filter, groupBy, join)
  - Lazy operations to build RDDs from other RDDs
- Actions (e.g. count, collect, save)
  - Return a result or write it to storage
Example: Mining Console Logs

Load error messages from a log into memory, then interactively search

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split(\'\t\')[2])
messages.persist()

messages.filter(lambda s: "foo" in s).count()
messages.filter(lambda s: "bar" in s).count()
...
```

**Result:** full-text search of Wikipedia in <1 sec  
(vs 20 sec for on-disk data)

**Result:** scaled to 1 TB data in 5-7 sec  
(vs 170 sec for on-disk data)
RDD Fault Tolerance

RDDs track the transformations used to build them (their *lineage*) to recompute lost data.

E.g:

```python
messages = textFile(...).filter(lambda s: s.contains("ERROR")).map(lambda s: s.split("\t")[2])
```
Lineage

- Spark uses the lineage to schedule jobs
  - Transformation on the same partition form a stage
    - Joins, for example, are a stage boundary
    - Need to reshuffle data
  - A job runs a single stage
    - pipeline transformation within a stage
  - Schedule job where the RDD partition is
Lineage & Fault Tolerance

• Great opportunity for *efficient* fault tolerance
  • Let's say one machine fails
  • Want to recompute only its state
  • The lineage tells us what to recompute
    • Follow the lineage to identify all partitions needed
    • Recompute them

• For last example, identify partitions of lines missing
  • All dependencies are “narrow”; each partition is dependent on one parent partition
  • Need to read the missing partition of lines; recompute the transformations
Fault Recovery

The diagram illustrates the iteration time (s) for various iterations. The iteration time increases with each iteration, reaching a peak at iteration 6. After iteration 6, the iteration time decreases. The diagram highlights that a failure occurs at iteration 6, indicated by the peak iteration time of 81 seconds.
Example: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each page’s rank to

$$\sum_{i \in \text{neighbors}} \frac{\text{rank}_i}{|\text{neighbors}_i|}$$

```scala
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
    ranks = links.join(ranks).flatMap {
        (url, (links, rank)) =>
        links.map(dest => (dest, rank/links.size))
    }.reduceByKey(_ + _)
}
```
Optimizing Placement

- links & ranks repeatedly joined
- Can co-partition them (e.g., hash both on URL)
- Can also use app knowledge, e.g., hash on DNS name
PageRank Performance

![Diagram showing time per iteration for Hadoop, Basic Spark, and Spark + Controlled Partitioning]
TensorFlow: System for ML

- Open Source, lots of developers, external contributors
- Used in: RankBrain (rank results), Photos (image recognition), SmartReply (automatic email responses)
Three types of ML

- Large scale training: huge datasets, generate models
  - Google’s previous DistBelief for 100s of machines
- Low latency inference: running models in datacenters, phones, etc.
  - Custom engines
- Testing new ideas
  - Single node flexible systems (Torch, Theano)
TensorFlow

- Common way to write programs
- Dataflow + Tensors
- Mutable state
- Simple mathematical operations
- Automatic differentiation
Background: NN Training

- Take input image
- Compute loss function (forward pass)
- Compute error gradients (backward pass)
- Update weights
- Repeat
Computation is a DFG

Graph of Nodes, also called Operations or ops.
import tensorflow as tf
a = tf.constant(2, name='a')
b = tf.constant(3, name='b')
A = tf.add(a, b, name = 'add')

with tf.Session() as sess:
    print(sess.run(A))
Example Code

```python
# 1. Construct a graph representing the model.
x = tf.placeholder(tf.float32, [BATCH_SIZE, 784])  # Placeholder for input.
y = tf.placeholder(tf.float32, [BATCH_SIZE, 10])  # Placeholder for labels.

W_1 = tf.Variable(tf.random_uniform([[784, 100]]))  # 784x100 weight matrix.
b_1 = tf.Variable(tf.zeros([100]))  # 100-element bias vector.
layer_1 = tf.nn.relu(tf.matmul(x, W_1) + b_1)  # Output of hidden layer.

W_2 = tf.Variable(tf.random_uniform([[100, 10]]))  # 100x10 weight matrix.
b_2 = tf.Variable(tf.zeros([10]))  # 10-element bias vector.
layer_2 = tf.matmul(layer_1, W_2) + b_2  # Output of linear layer.

# 2. Add nodes that represent the optimization algorithm.
loss = tf.nn.softmax_cross_entropy_with_logits(layer_2, y)  # Connect to the TF runtime.
train_op = tf.train.AdagradOptimizer(0.01).minimize(loss)  # Randomly initialize weights.

# 3. Execute the graph on batches of input data.
with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())  # Train iteratively for NUM_STEPS.
    for step in range(NUM_STEPS):
        x_data, y_data = ...  # Load one batch of input data.
        sess.run(train_op, {x: x_data, y: y_data})  # Perform one training step.
```
Parameter Server Architecture

Stateless workers, stateful parameter servers (DHT)
Commutative updates to parameter server
TensorFlow

- Flexible architecture for mapping operators and parameter servers to different devices
- Supports multiple concurrent executions on overlapping subgraphs of the overall graph
- Individual vertices may have mutable state that can be shared between different executions of the graph
TensorFlow handles the glue
Asynchronous execution is sometimes helpful, addresses stragglers

Asynchrony causes consistency problems

TensorFlow: pursues synchronous training
  - But adds k backup machines to reduce the straggler problem
  - Uses domain specific knowledge to enable this optimization
Open Research Problems

- Automatic placement: data flow - great mechanism, but not clear how to use it appropriately
- Mutable state - split round-robin across parameter server nodes, stateless tasks replicated on GPUs as much as it fits, rest on CPUs
- How to take data flow representation to generate more efficient code?