Motivation

• Large datasets are inexpensive to collect, but require high parallelism to process
  – 1 TB of disk space = $50
  – Reading 1 TB from disk = 6 hours
  – Reading 1 TB from 1000 disks = 20 seconds

• Not only queries, but all computations will need to scale (data loading, complex analytics, etc)

• How do we program large clusters?
Traditional Network Programming

- Message-passing between nodes
- Very difficult to use at scale:
  - How to split problem across nodes?
    - Must consider network, data locality
  - How to deal with failures?
    - 1 server fails every 3 years => 10K nodes see 10 faults per day
    - Even worse: stragglers (node is not failed, but slow)

Almost nobody directly uses this model!

Data-Parallel Models

- Restrict the programming interface so that the system can do more automatically
- "Here's an operation, run it on all of the data"
  - I don't care where it runs (you schedule that)
  - In fact, feel free to run it twice on different nodes
- Biggest example: MapReduce
Software in this Space

Hadoop  Dryad  Spark
Hive    Pig    DryadLINQ  Shark
Pregel  Giraph Impala  Tez
GraphLab  Dremel  Storm
Giraph  Impala  Tez
GraphLab  Dremel  Storm
Giraph  Impala  Tez

Applications

• Extract, transform and load ("ETL")
• Web indexing (Google)
• Spam filtering (Yahoo!)
• Product recommendation (Netflix)
• Ad-hoc queries (Facebook)
• Fraud detection

This Lecture

• Challenges in large-scale environments
• MapReduce model
• Limitations and extensions of MapReduce
• Other types of platforms
Tackling The Challenges of Big Data

Big Data Storage
Distributed Computing Platforms

Introduction

THANK YOU
**Goals of Large-Scale Data Platforms**

- **Scalability in number of nodes:**
  - Parallelize I/O to quickly scan large datasets

- **Cost-efficiency:**
  - Commodity nodes (cheap, but unreliable)
  - Commodity network (low bandwidth)
  - Automatic fault-tolerance (fewer admins)
  - Easy to use (fewer programmers)

---

**Typical Hadoop Cluster**

- 40 nodes/rack, 1000-4000 nodes in cluster
- 10 Gbps bandwidth in rack, 40 Gbps out of rack
- Node specs (Cloudera):
  - 8-16 cores, 64-512 GB RAM, 12×2 TB disks
Challenges of Cluster Environment

- Cheap nodes fail, especially when you have many
  - Mean time between failures for 1 node = 3 years
  - MTBF for 1000 nodes = 1 day
  - Solution: Build fault tolerance into system

- Commodity network = low bandwidth
  - Solution: Push computation to the data

- Programming distributed systems is hard
  - Solution: Data-parallel model: users write map/reduce functions, system handles work distribution and faults

Typical Software Components

- Distributed file system (e.g. Hadoop’s HDFS)
  - Single namespace for entire cluster
  - Replicates data 3x for fault-tolerance

- MapReduce system (e.g. Hadoop MapReduce)
  - Runs jobs submitted by users
  - Manages work distribution & fault-tolerance
  - Colocated with file system

Hadoop Distributed File System

- Files split into blocks
- Blocks replicated across several nodes (often 3)
- Master stores metadata (file names, locations, …)
- Optimized for large files, sequential reads
Tackling The Challenges of Big Data
Big Data Storage
Distributed Computing Platforms
Large-Scale Computing Environments

THANK YOU

Matei Zaharia
Assistant Professor
Massachusetts Institute of Technology

Introduction to MapReduce

Matei Zaharia
Assistant Professor
Massachusetts Institute of Technology
MapReduce Programming Model

- A MapReduce job consists of two user functions
- Both operate on key-value records

- Map function:
  \[(K_{\text{in}}, V_{\text{in}}) \rightarrow \text{list}(K_{\text{inter}}, V_{\text{inter}})\]

- Reduce function:
  \[(K_{\text{inter}}, \text{list}(V_{\text{inter}})) \rightarrow \text{list}(K_{\text{out}}, V_{\text{out}})\]

Example: Word Count

def map(line):
    for word in line.split():
        output(word, 1)

def reduce(key, values):
    output(key, sum(values))

Word Count Execution
MapReduce Execution

- Automatically split work into many small tasks
- Send map tasks to nodes based on data locality
  - Typically, files are replicated so that there are 3 copies of each block
- Load-balance dynamically as tasks finish

Fault Recovery

1. If a task crashes:
   - Retry on another node
   *OK for a map because it had no dependencies
   *OK for reduce because map outputs are on disk
   - If the same task repeatedly fails, end the job

   Requires user code to be deterministic and idempotent

Fault Recovery

2. If a node crashes:
   - Relaunch its current tasks on other nodes
   - Relaunch any maps the node previously ran
   *Necessary because their output files were lost along with the crashed node
**Fault Recovery**

3. If a task is going slowly (straggler):
   - Launch second copy of task on another node
   - Take the output of whichever copy finishes first, and cancel the other one

By offering a data-parallel model, MapReduce can handle many distribution issues automatically.
1. Search

- Input: (lineNumber, line) records
- Output: lines matching a given pattern

- Map:
  \[
  \text{if (line matches pattern): output(line, lineNumber)}
  \]

- Reduce: identity function
  - Alternative: no reducer (map-only job)

2. Sort

- Input: (key, value) records
- Output: same records, sorted by key

- Map: identity function
- Reduce: identity function

- Pick partitioning function \( p \) so that \( k_1 < k_2 \implies p(k_1) < p(k_2) \)
3. Inverted Index

- Input: (filename, text) records
- Output: list of files containing each word

- Map:
  ```python
  for word in text.split():
    output(word, filename)
  ```

- Reduce:
  ```python
  def reduce(word, filenames):
    output(word, unique(filenames))
  ```

Inverted Index Example

- to, hamlet.txt
- be, hamlet.txt
- not, hamlet.txt
- or, hamlet.txt
- afraid, 12th.txt
- to be or not to be
- greatness, 12th.txt
- be not afraid of greatness

4. Most Popular Words

- Input: (filename, text) records
- Output: the 100 words occurring in most files

- Two-stage solution:
  - Job 1:
    - Create inverted index, giving (word, list(file)) records
  - Job 2:
    - Map each (word, list(file)) to (count, word)
    - Sort these records by count as in sort job
Tackling The Challenges of Big Data

Big Data Storage
Distributed Computing Platforms
MapReduce Examples

THANK YOU

Tackling The Challenges of Big Data

Big Data Storage

Matei Zaharia
Assistant Professor
Massachusetts Institute of Technology

Tackling The Challenges of Big Data

Big Data Storage
Distributed Computing Platforms
Models Built on MapReduce

Matei Zaharia
Assistant Professor
Massachusetts Institute of Technology
Limitations of MapReduce

- MapReduce is great at single-pass analysis, but most applications require multiple MR steps

- Examples:
  - Google indexing pipeline: 21 steps
  - Analytics queries (e.g. sessions, top K): 2-5 steps
  - Iterative algorithms (e.g. PageRank): 10’s of steps

- Two problems: programmability & performance

Programmability

- Multi-step jobs lead to convoluted code
  - 21 MR steps -> 21 mapper and reducer functions

- Repeated code for common patterns

Performance

- MR only provides one pass of computation
  - Must write out data to file system in-between jobs

- Expensive for apps that need to reuse data
  - Multi-pass algorithms (e.g. PageRank)
  - Interactive data mining (many queries on same data)
Reactions

• Higher-level interfaces over MapReduce
  – Translate a higher-level language into MR steps
  – May merge steps to optimize performance
  – Examples: Hive, Pig

• Generalizations of the model
  – Examples: Dryad, Spark

Hive

• Relational data warehouse over Hadoop
  – Maintains a catalog of tables with schemas
  – Runs queries in a subset of SQL
  – Supports traditional query optimization, as well as complex data types and MapReduce scripts

• Developed at Facebook
• Used for most of Facebook’s MR jobs

Hive Example

• Given a tables of page views and user info, find top 5 pages visited by users under 20

  SELECT url, COUNT(*) AS views
  FROM users u, pageviews v
  WHERE u.name = v.username
  AND u.age < 20
  ORDER BY views DESC
  LIMIT 5

Example from https://cwiki.apache.org/confluence/download/attachments/26805800/HadoopSummit2009.ppt
Pig Example

Users = load 'users.txt' as (name, age);
Filtered = filter Users by age < 20;
Pages = load 'pageviews.txt' as (user, url);
Joined = join Filtered by name, Pages by user;
Grouped = group Joined by url;
Summed = foreach Grouped generate group, count(Joined) as clicks;
Sorted = order Summed by clicks desc;
Top5 = limit Sorted 5;
store Top5 into 'top5sites';

• Scripting-like language offering SQL operators
• Developed and widely used at Yahoo!

Example from https://cwiki.apache.org/confluence/download/attachments/26805800/HadoopSummit2009.ppt

Tackling The Challenges of Big Data
Big Data Storage
Distributed Computing Platforms
Models Built on MapReduce

THANK YOU
Motivation

- Compiling high-level operators to MapReduce helps, but multi-step apps still have limitations
  - Sharing data between MapReduce jobs is slow (requires writing to replicated file system)
  - Some apps need to do multiple passes over same data

- E.g., word count + sort:

Dryad Model

- General graphs of tasks instead of two-level MapReduce graph

- Similar recovery mechanisms (replay parent tasks in graph to recover, replicate slow tasks)

[Isard et al, EuroSys 2007]
Spark Model

- Dryad supports richer operator graphs, but data flow is still acyclic
- Many applications need to efficiently reuse data
  - Interactive data mining: run multiple user queries (not known in advance) on same subset of data
  - Iterative algorithms: make multiple passes over data

Spark Model

- Let users explicitly build and persist distributed datasets
- Key idea: Resilient Distributed Datasets (RDDs)
  - Collections of objects partitioned across cluster that can be stored on disk or in memory
  - Built through graphs of parallel transformations (e.g. map, reduce, group-by)
  - Automatically rebuilt on failure
- High-level APIs in Java, Scala, Python

Example: Log Mining

Load error messages from a log file in memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda x: x.startswith("ERROR")
messages = errors.map(lambda x: x.split('t')[2])
messages.persist()
messages.filter(lambda x: "foo" in x).count()
messages.filter(lambda x: "bar" in x).count()
```

Result: full-text search of Wikipedia in 1 sec (vs 30 sec for on-disk data)
Fault Recovery

RDDs track their lineage graph to rebuild lost data

```
file.map(lambda rec: (rec.type, 1))
  .reduceByKey(lambda x, y: x + y)
  .filter(lambda (type, count): count > 10)
```

Benefits of RDDs

- Can express general computation graphs, similar to Dryad
- No need to replicate data for fault tolerance
  - Write at memory speed instead of network speed
  - Use less memory than replicated systems
- Persisting is only a hint
  - If there's too little memory, can spill to disk or just drop old data
Iterative Algorithms

K-means Clustering

<table>
<thead>
<tr>
<th>Time (sec)</th>
<th>Hadoop MR</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4.1</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Logistic Regression

<table>
<thead>
<tr>
<th>Time (sec)</th>
<th>Hadoop MR</th>
<th>Spark</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.96</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tackling The Challenges of Big Data
Big Data Storage
Distributed Computing Platforms
Generalizations of MapReduce

THANK YOU

Tackling The Challenges of Big Data
Big Data Storage

Matei Zaharia
Assistant Professor
Massachusetts Institute of Technology
Tackling The Challenges of Big Data

Big Data Storage

Distributed Computing Platforms

Spark Demo

Matei Zaharia
Assistant Professor
Massachusetts Institute of Technology

THANK YOU
Introduction

• We’ve covered MapReduce and its extensions, which are the most popular models today

• This area is still seeing rapid change & innovation

• We’ll briefly sketch some related questions:
  - How do these efforts relate to databases and SQL?
  - What are some very different models from MapReduce?
  *Asynchronous computation, streaming

1. Convergence with SQL

• One of the first things users wanted to run in MapReduce clusters was SQL!

• Some systems build SQL engines over MapReduce cluster architectures
  - Google Dremel, Cloudera Impala, Apache Drill & Tez
  - Usually lack fault tolerance but target short queries

• Others implement many of the optimizations in database engines on MapReduce-like runtimes
  - Google Tenzing, Shark (Hive on Spark)
How This Works

- Several things make analytical databases fast
  - Efficient storage format (e.g. column-oriented)
  - Precomputation (e.g. indices, partitioning, statistics)
  - Query optimization
- Shark (Hive on Spark)
  - Implements column-oriented storage and processing within Spark records
  - Supports data partitioning and statistics on load
- Result: can also combine SQL with Spark code
  - From Spark: rdd = shark.sqlRdd("select * from users")
  - From SQL: generate KMeans(select lat, long from tweets)

In the Other Direction

- Some databases adding support for MapReduce
- Greenplum, Aster Data: MapReduce executes within database engine, on relational data
- Hadapt: hybrid database / MapReduce system
  - Queries sent to Hadoop cluster with DB on each node
  - Supports "schema on read" functionality (e.g. run SQL over JSON records)

2. Asynchronous Computing

- MapReduce & related models use deterministic, synchronized computation for fault recovery
- Some applications (e.g. numerical optimization, machine learning) can converge without this
  - Each step makes "progress" towards a goal
  - Losing state just moves you slightly away from goal
- Asynchronous systems use this to improve speed
  - Don't wait for all nodes to advance / communicate
  - Examples: GraphLab, Hogwild, Google DistBelief
3. Stream Processing

- Processing data in real-time can require very different designs

- **Example: Storm (continuous, stateful operators)**
  - Build graph of operators
  - System plays each record through at least once in case of failures

- **Several options for determinism on failures**
  - Transactional storage (Trident)
  - Run as sequence of batch jobs (Spark Streaming)

Resource Sharing

- Given all these programming models, how do we get them to coexist

- Cross-framework resource managers allow dynamically sharing a cluster between them
  - Let applications launch work through common API
  - Hadoop YARN, Apache Mesos
Conclusion

• Large-scale cluster environments are difficult to program with traditional methods
• Result is a profusion of new programming models that aim to simplify this
• Main idea: capture computation in a declarative manner to let system handle distribution

Tackling The Challenges of Big Data

Big Data Storage
Distributed Computing Platforms
Other Platforms

THANK YOU

Tackling The Challenges of Big Data

Big Data Storage
Distributed Computing Platforms

Matei Zaharia
Assistant Professor
Massachusetts Institute of Technology