MapReduce

Parallel computing platform built at Google

Still runs millions of jobs / day

“Functional” API with deterministic recomputation for fault tolerance
Key Ideas in MapReduce

Recomputation for fault tolerance

Parallel recovery: lost work is spread out

Straggler mitigation through backup tasks

Dynamic scheduling
Key Design Elements

Centralized master

“Pull” based communication model
– Reduce tasks fetch files from mappers
– Provides cheaper fault recovery and room for dynamic scheduling of tasks
Real-World MR Use Cases

Extract, Transform and Load (ETL)

SQL-like queries (Tenzing, Hive)

Complex analytics with non-SQL code
Spark

Generalizes MapReduce while retaining its scheduling and fault tolerance benefits

Main addition: efficient data sharing

Enables more applications
- Iterative algorithms
- Interactive queries
- Stream processing
Resilient Distributed Datasets (RDDs)

Restricted form of shared memory
- Immutable, partitioned sets of records
- Can only be built through coarse-grained, deterministic operations (map, filter, join, …)

Fault recovery using *lineage*
- Log one operation to apply to many elements
- Recompute lost partitions on failure

[NSDI 2012]
RDD Recovery

Input file

map($f$)  group-by($g$)  filter($h$)
RDD Recovery

map(f)  group-by(g)  filter(h)

Input file
RDD Recovery

map(f)  group-by(g)  filter(h)

Input file
Tradeoff Space

Granularity of Updates

- Fine
- Coarse

Write Throughput

- Low
- High

Network bandwidth

- Memory bandwidth

K-V stores, databases

GFS

RDDs
<table>
<thead>
<tr>
<th>Aspect</th>
<th>RDDs</th>
<th>Dist. Shared Mem. (including key-value stores, etc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writes</td>
<td>Coarse-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Reads</td>
<td>Fine-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Consistency</td>
<td>Trivial (immutable)</td>
<td>Expensive</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Fine-grained &amp; low-cost using lineage</td>
<td>Replication or checkpoint/rollback</td>
</tr>
<tr>
<td>Straggler recovery</td>
<td>Possible using speculation</td>
<td>Difficult</td>
</tr>
</tbody>
</table>
Other Differences from MR

1. Explicit partitioning, partitioning-aware ops
   - E.g. a 3x speedup in PageRank

2. More complex DAGs of tasks
   - Better performance even if data is not reused
# RDD API

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>partitions()</td>
<td>Return a list of Partition objects</td>
</tr>
<tr>
<td>preferredLocations(p)</td>
<td>List nodes where partition p can be accessed faster due to data locality</td>
</tr>
<tr>
<td>dependencies()</td>
<td>Return a list of dependencies</td>
</tr>
<tr>
<td>iterator(p, parentIters)</td>
<td>Compute the elements of partition p given iterators for its parent partitions</td>
</tr>
<tr>
<td>partitioner()</td>
<td>Return metadata specifying how RDD records are partitioned across nodes</td>
</tr>
</tbody>
</table>
Supported Applications

Iterative MapReduce (e.g. machine learning)
Pregel-like graph processing
Interactive ad-hoc queries
More were built later (e.g. SQL, streaming)
How General is Spark?

*MapReduce + data sharing can emulate any distributed system!*

- Local computation
- All-to-all communication

One MR step

How to share data quickly across steps?
Spark: RDDs

How big is this latency?
Spark: ~100 ms
# Push vs Pull-Based Systems

“Push” = senders write to receivers (e.g. parallel DB)  
“Pull” = senders write locally, receivers fetch (e.g. MR)

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Push</th>
<th>Pull</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>Lower</td>
<td>Higher</td>
</tr>
<tr>
<td>Throughput</td>
<td>Similar</td>
<td>Similar</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Expensive (rerun all senders)</td>
<td>Cheap</td>
</tr>
<tr>
<td>Straggler recovery</td>
<td>Difficult</td>
<td>Easy (backup tasks)</td>
</tr>
<tr>
<td>Elasticity / multitenancy</td>
<td>Difficult</td>
<td>Easy</td>
</tr>
</tbody>
</table>