Lecture 12:

Cluster Computing
(MapReduce and Spark)

Parallel Computing
Stanford CS149, Winter 2019
Which program performs better?

**Program 1**

```c
void add(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] * B[i];
}

// assume arrays are allocated here

// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

**Program 2**

```c
void fused(int n, float* A, float* B, float* C, float* D, float* E) {
    for (int i=0; i<n; i++)
        E[i] = D[i] + (A[i] + B[i]) * C[i];
}

// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```

The transformation of the code in program 1 to the code in program 2 is called “loop fusion”
Example involves globally restructuring the order of computation to improve producer-consumer locality

(improve arithmetic intensity of program)
Commodity Clusters

- **Standard architecture:**
  - Cluster of commodity Linux servers (multicore x86)
  - Private memory ⇒ separate address spaces
  - Ethernet network ⇒ (relatively) low bandwidth

- **Inexpensive**
  - Built from commodity processors, networks & storage
  - 1000s of nodes for < $10M

- **How to organize computations on this architecture?**
  - Mask issues such as load balancing and failures
Warehouse Size Cluster
Typical Commodity Server

RAM (64-512GB)

CPU x2 (8-64 cores)

Disks x10 (10-30TB)

50 GB/s

Network

1-4 GB/s

50 MB/s each

Top-of-Rack Switch

Nodes in Same Rack

1-4 GB/s

Nodes in Other Racks

0.1-4 GB/s

~40 1RU serves per rack
Using A Cluster

- Want to process 100TB (1 day Facebook data, $4000 to store!)
- On 1 node: scanning @ 50MB/s = 23 days
- On 1000 nodes: scanning @ 50MB/s = 33 min
- But, very hard to utilize 1000 nodes!
  - Hard to program 8,000 cores
  - Something breaks every hour
  - Need efficient, reliable and usable framework
Cluster Storage Systems

- First order problem: if nodes can fail, how can we store data persistently?

- Answer: distributed storage systems
  - Provide global namespace
  - Examples: Google GFS, Hadoop HDFS, Amazon S3

- Typical usage patterns
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common (e.g. log files)
Distributed File Systems

- **Chunk servers**
  - a.k.a. DataNodes in HDFS
  - File is split into contiguous chunks (usually 64-256 MB)
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks

- **Master node**
  - a.k.a. NameNode in HDFS
  - Stores metadata; might be replicated

- **Client library for file access**
  - Talks to master to find chunk (data) servers
  - Connects directly to chunk servers to access data
Example: Hadoop Distributed FS (HDFS)

- Global namespace
- Files split into ~200MB blocks
- Each block replicated on multiple DataNodes
- Intelligent client
  - Finds locations of blocks from NameNode; requests data from DataNode
Cluster Computing Frameworks

- Want frameworks to automatically handle:
  - Work placement (e.g. data locality)
  - Load balancing
  - Communication
  - Failures & stragglers

- Examples: MapReduce, Dryad, Spark, etc
First Major Example: MapReduce

- Data type: key-value records

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

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

Input
- the quick brown fox
- the fox ate the mouse
- how now brown cow

Map
- the, 1
- brown, 1
- fox, 1
- the, 1
- quick, 1
- the, 1

Shuffle & Sort
- the, 1
- brown, 1
- fox, 1
- the, 1
- quick, 1
- how, 1
- now, 1
- brown, 1
- ate, 1
- mouse, 1
- cow, 1

Reduce
- brown, 2
- fox, 2
- how, 1
- now, 1
- the, 3
- ate, 1
- cow, 1
- mouse, 1
- quick, 1

Output
Benefits

- By providing a functional API, MapReduce greatly simplified cluster programming:
  - Automatic division of job into tasks
  - Locality-aware scheduling
  - Load balancing
  - Recovery from failures & stragglers
Let’s say CS149 gets very popular...
The log of page views gets quite large…

Assume cs149log.txt is a large file, stored in a distributed file system, like HDFS

Below: cluster of 4 nodes, each node with a 1 TB disk
Contents of cs149log.txt are distributed evenly in blocks across the cluster
Imagine your professors want to know a bit more about the glut of students visiting the CS149 web site...

For example:
“What type of mobile phone are all these students using?”
Consider a simple programming model

```java
void mapper(string line, multimap<string,string>& results) {
    string user_agent = parse_requester_user_agent(line);
    if (is_mobile_client(user_agent))
        results.add(user_agent, 1);
}

void reducer(string key, list<string> values, int& result) {
    int sum = 0;
    for (v in values)
        sum += v;
    result = sum;
}
```

```java
LineByLineReader input("hdfs://cs149log.txt");
Writer output("hdfs://…");
runMapReduceJob(mapper, reducer, input, output);
```

(The code above computes the count of page views by each type of mobile phone.)
Let’s design an implementation of runMapReduceJob
Step 1: running the mapper function

```c++
// called once per line in file
void mapper(string line, multimap<string,string>& results) {
    string user_agent = parse_requester_user_agent(line);
    if (is_mobile_client(user_agent))
        results.add(user_agent, 1);
}

// called once per unique key in results
void reducer(string key, list<string> values, int& result) {
    int sum = 0;
    for (v in values)
        sum += v;
    result = sum;
}
```

LineByLineReader input("hdfs://cs149log.txt");
Writer output("hdfs://...");
runMapReduceJob(mapper, reducer, input, output);

---

Step 1: run mapper function on all lines of file

**Question:** How to assign work to nodes?

**Idea 1:** use work queue for list of input blocks to process
Dynamic assignment: free node takes next available block

**Idea 2:** data distribution based assignment: Each node processes lines in blocks of input file that are stored locally.

---

![Diagram of nodes and blocks]

Node 0
- CPU
- Disk
  - cs149log.txt
  - block 0

Node 1
- CPU
- Disk
  - cs149log.txt
  - block 1

Node 2
- CPU
- Disk
  - cs149log.txt
  - block 2

Node 3
- CPU
- Disk
  - cs149log.txt
  - block 3

...
### Steps 2 and 3: gathering data, running the reducer

```cpp
void mapper(string line, map<string,string> results) {
    string user_agent = parse_requester_user_agent(line);
    if (is_mobile_client(user_agent))
        results.add(user_agent, 1);
}

void reducer(string key, list<string> values, int& result) {
    int sum = 0;
    for (v in values)
        sum += v;
    result = sum;
}
```

Step 2: Prepare intermediate data for reducer

Step 3: Run reducer function on all keys

Question: how to assign reducer tasks?

Question: how to get all data for key onto the correct worker node?

Keys to reduce:
(generated by mapper):
- Safari iOS
- Chrome
- Safari iWatch
- Chrome Glass

R reducer tasks
\[
\text{task} = \text{hash(key)} \mod R
\]
Steps 2 and 3: gathering data, running the reducer

```
// gather all input data for key, then execute reducer
// to produce final result
void runReducer(string key, reducer, result) {
    list<string> inputs;
    for (n in nodes) {
        filename = get_filename(key, n);
        read lines of filename, append into inputs;
    }
    reducer(key, inputs, result);
}
```

Step 2: Prepare intermediate data for reducer.
Step 3: Run reducer function on all keys.
Question: how to assign reducer tasks?
Question: how to get all data for key onto the correct worker node?

Keys to reduce:
(generated by mapper):
- Safari iOS
- Chrome
- Safari iWatch
- Chrome Glass

Example:
 Assign Safari iOS to Node 0

Node 0
- CPU
- Disk
  - Safari iOS values 0
  - Chrome values 0
  - cs149log.txt block 0

Node 1
- CPU
- Disk
  - Safari iOS values 1
  - Chrome values 1
  - cs149log.txt block 2

Node 2
- CPU
- Disk
  - Safari iOS values 2
  - Chrome values 2
  - cs149log.txt block 4

Node 3
- CPU
- Disk
  - Safari iOS values 3
  - Chrome values 3
  - cs149log.txt block 6
MapReduce Execution Summary

- map() reduce()

1. Partition input key/value pairs into chunks, run map() tasks in parallel
2. After all map()s are complete, consolidate all emitted values for each unique emitted key
3. Now partition space of output map keys, and run reduce() in parallel
Additional implementation challenges at scale

Nodes may fail during program execution

Some nodes may run slower than others (due to different amounts of work, heterogeneity in the cluster, etc..)
Job scheduler responsibilities

- **Exploit data locality: “move computation to the data”**
  - Run mapper jobs on nodes that contain input files
  - Run reducer jobs on nodes that already have most of data for a certain key

- **Handling node failures**
  - Scheduler detects job failures and reruns job on new machines
    - This is possible since inputs reside in persistent storage (distributed file system)
  - Scheduler duplicates jobs on multiple machines (reduce overall processing latency incurred by node failures)

- **Handling slow machines**
  - Scheduler duplicates jobs on multiple machines
runMapReduceJob problems?

- Permits only a very simple program structure
  - Programs must be structured as: map, followed by reduce by key
  - See DryadLINQ for generalization to DAGs

- Iterative algorithms must load from disk each iteration
  - Example of graph processing (page rank):
    ```c
    void pagerank_mapper(graphnode n, map<string,string> results) {
      float val = compute update value for n
      for (dst in outgoing links from n)
        results.add(dst.node, val);
    }
    
    void pagerank_reducer(graphnode n, list<float> values, float& result) {
      float sum = 0.0;
      for (v in values)
        sum += v;
      result = sum;
    }
    
    for (i = 0 to NUM_ITERATIONS) {
      input = load graph from last iteration
      output = file for this iteration output
      runMapReduceJob(pagerank_mapper, pagerank_reducer, result[i-1], result[i]);
    }
    ```
Problems with MapReduce

1. **Performance**
   - No primitives for data sharing between jobs!
   - Need to go to distributed file system each read/write

2. **Programmability**
   - 10s of map/reduce functions create spaghetti code
   - API does not provide type safety
in-memory, fault-tolerant distributed computing
http://spark.apache.org/

[Zaharia et al. NSDI 2012]
Goals

- Programming model for cluster-scale computations where there is significant reuse of intermediate datasets
  - Iterative machine learning and graph algorithms
  - Interactive data mining: load large dataset into aggregate memory of cluster and then perform multiple ad-hoc queries

- Don’t want incur inefficiency of writing intermediates to persistent distributed file system (want to keep it in memory)
  - Challenge: efficiently implementing fault tolerance for large-scale distributed in-memory computations.
Fault tolerance for in-memory calculations

- Replicate all computations
  - Expensive solution: decreases peak throughput

- Checkpoint and rollback
  - Periodically save state of program to persistent storage
  - Restart from last checkpoint on node failure

- Maintain log of updates (commands and data)
  - High overhead for maintaining logs

Recall map-reduce solutions:
- Checkpoints after each map/reduce step by writing results to file system
- Scheduler’s list of outstanding (but not yet complete) jobs is a log
- Functional structure of programs allows for restart at granularity of a single mapper or reducer invocation (don’t have to restart entire program)
Spark Programming Model

Resilient distributed datasets (RDDs)
- Immutable, partitioned collections of objects
- Created through parallel transformations (map, filter, groupBy, join, ...) on data in stable storage
- Can be cached for efficient reuse

Actions on RDDs
- Count, reduce, collect, save, ...
- Generate result on master
Creating RDDs

// Turn a Scala collection into an RDD
spark.parallelize(List(1, 2, 3))

// Load text file from local FS, HDFS, or S3
spark.textFile("file.txt")
spark.textFile("directory/*.txt")
spark.textFile("hdfs://namenode:9000/path/file")
Basic Transformations

// Pass each element through a function
val squares = nums.map(x => x*x) // {1, 4, 9}

// Keep elements passing a predicate
val even = squares.filter(x => x % 2 == 0) // {4}

// Map each element to zero or more others
nums.flatMap(x => 1 to x) // => {1, 1, 2, 1, 2, 3}
## Basic RDD Actions

```scala
val nums = spark.parallelize(List(1, 2, 3))

// Retrieve RDD contents as a local collection
collect() // => Array(1, 2, 3) could be too big!
// Return first K elements
take(2) // => Array(1, 2)

// Count number of elements
count() // => 3

// Merge elements with an associative function
reduce((a, b) => a + b) // => 6

// Write elements to a text file
saveAsTextFile("hdfs://file.txt")
```
Working with Key-Value Pairs

- Spark’s “distributed reduce” transformations operate on RDDs of key-value pairs

- Scala pair syntax:
  ```scala
  val pair = (a, b) // sugar for new Tuple2(a, b)
  ```

- Accessing pair elements:
  ```scala
  pair._1 // => a
  pair._2 // => b
  ```
Some Key-Value Operations

```scala
val pets = sc.parallelize(List(("cat", 1), ("dog", 1), ("cat", 2)))
pets.reduceByKey(_ + _) // => {(cat, 3), (dog, 1)}
pets.groupByKey() // => {(cat, Seq(1, 2)), (dog, Seq(1))}
pets sortByKey() // => {(cat, 1), (cat, 2), (dog, 1)}
```

- `reduceByKey` also locally combines on map side
Example: Word Count

```scala
val lines = spark.textFile("hamlet.txt")
val counts = lines.flatMap(line => line.split(" "))
  .map(word => (word, 1))
  .reduceByKey(_ + _)
```

```

```

```

```

```

```

```

```

```

```

```

```

```

```
Resilient distributed dataset (RDD)

Spark’s key programming abstraction:

- Read-only ordered collection of records (immutable)
- RDDs can only be created by deterministic transformations on data in persistent storage or on existing RDDs
- Actions on RDDs return data to application

```scala
// create RDD from file system data
var lines = spark.textFile("hdfs://cs149log.txt");

// create RDD using filter() transformation on lines
var mobileViews = lines.filter((x: String) => isMobileClient(x));

// another filter() transformation
var safariViews = mobileViews.filter((x: String) => x.contains("Safari"));

// then count number of elements in RDD via count() action
var numViews = safariViews.count();
```
Repeating the map-reduce example

// 1. create RDD from file system data
// 2. create RDD with only lines from mobile clients
// 3. create RDD with elements of type (String,Int) from line string
// 4. group elements by key
// 5. call provided reduction function on all keys to count views
var perAgentCounts = spark.textFile("hdfs://cs149log.txt")
    .filter(x => isMobileClient(x))
    .map(x => (parseUserAgent(x),1));
    .reduceByKey((x,y) => x+y)
    .collect();
Another Spark program

// create RDD from file system data
var lines = spark.textFile("hdfs://cs149log.txt");

// create RDD using filter() transformation on lines
var mobileViews = lines.filter((x: String) => isMobileClient(x));

// instruct Spark runtime to try to keep mobileViews in memory
mobileViews.persist();

// create a new RDD by filtering mobileViews
// then count number of elements in new RDD via count() action
var numViews = mobileViews.filter(_.contains("Safari")).count();

// 1. create new RDD by filtering only Chrome views
// 2. for each element, split string and take timestamp of
//    page view
// 3. convert RDD to a scalar sequence (collect() action)
var timestamps = mobileViews.filter(_.contains("Chrome"))
    .map(_.split(" ")(0))
    .collect();
RDD transformations and actions

Transformations: (data parallel operators taking an input RDD to a new RDD)

- **map**: $f : T \Rightarrow U$ → $\text{RDD}[T] \Rightarrow \text{RDD}[U]
- **filter**: $f : T \Rightarrow \text{Bool}$ → $\text{RDD}[T] \Rightarrow \text{RDD}[T]
- **flatMap**: $f : T \Rightarrow \text{Seq}[U]$ → $\text{RDD}[T] \Rightarrow \text{RDD}[U]
- **sample**: $\text{fraction : Float}$ → $\text{RDD}[T] \Rightarrow \text{RDD}[T]$ (Deterministic sampling)
- **groupByKey**: $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, \text{Seq}[V])]$
- **reduceByKey**: $f : (V, V) \Rightarrow V$ → $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$
- **union**: $\text{RDD}[T], \text{RDD}[T] \Rightarrow \text{RDD}[T]$
- **join**: $\text{RDD}[(K, V)], \text{RDD}[(K, W)] \Rightarrow \text{RDD}[(K, (V, W))]$
- **cogroup**: $\text{RDD}[(K, V)], \text{RDD}[(K, W)] \Rightarrow \text{RDD}[(K, (\text{Seq}[V], \text{Seq}[W]))]$
- **crossProduct**: $\text{RDD}[T], \text{RDD}[U] \Rightarrow \text{RDD}[(T, U)]$
- **mapValues**: $f : V \Rightarrow W$ → $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, W)]$ (Preserves partitioning)
- **sortBy**: $c : \text{Comparator}[K]$ → $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$
- **partitionBy**: $p : \text{Partitioner}[K]$ → $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$

Actions: (provide data back to the “host” application)

- **count**: $\text{RDD}[T] \Rightarrow \text{Long}$
- **collect**: $\text{RDD}[T] \Rightarrow \text{Seq}[T]$
- **reduce**: $f : (T, T) \Rightarrow T$ → $\text{RDD}[T] \Rightarrow T$
- **lookup**: $k : K$, $\text{RDD}[(K, V)] \Rightarrow \text{Seq}[V]$ (On hash/range partitioned RDDs)
- **save**: $\text{path : String}$ → Outputs RDD to a storage system, e.g., HDFS

Very similar to Scala Collections API
How do we implement RDDs?
In particular, how should they be stored?

```javascript
var lines = spark.textFile("hdfs://cs149log.txt");
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```

**Question:** should we think of RDD’s like arrays?
How do we implement RDDs?
In particular, how should they be stored?

val lines = spark.textFile("hdfs://cs149log.txt");
val lower = lines.map(_.toLowerCase);
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();

In-memory representation would be huge! (larger than original file on disk)
RDD partitioning and dependencies

```scala
val lines = spark.textFile("hdfs://cs149log.txt");
val lower = lines.map(_.toLower());
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();
```

Black lines show dependencies between RDD partitions.

Stanford CS149, Winter 2019
Implementing sequence of RDD ops efficiently

```scala
val lines = spark.textFile("hdfs://cs149log.txt");
val lower = lines.map(_.toLower());
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();
```

Recall “loop fusion” examples from opening slides of lecture

The following code stores only a line of the log file in memory, and only reads input data from disk once ("streaming" solution)

```java
int count = 0;
while (inputFile.eof()) {
    string line = inputFile.readLine();
    string lower = line.toLower;
    if (isMobileClient(lower))
        count++;
}
```
A simple interface for RDDs

```scala
val lines = spark.textFile("hdfs://cs149log.txt");
val lower = lines.map(_.toLower());
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();
```

// create RDD by mapping map_func onto input (parent) RDD
RDD::map(RDD parent, map_func) {
    return new RDDFromMap(parent, map_func);
}

// create RDD by filtering input (parent) RDD
RDD::filter(RDD parent, filter_func) {
    return new RDDFromFilter(parent, filter_func);
}

// create RDD from text file on disk
RDD::textFile(string filename) {
    return new RDDFromTextFile(open(filename));
}

// count action (forces evaluation of RDD)
RDD::count() {
    int count = 0;
    while (hasMoreElements()) {
        var el = next();
        count++;
    }
}
```
Narrow dependencies

“Narrow dependencies” = each partition of parent RDD referenced by at most one child RDD partition
- Allows for fusing of operations (here: can apply map and then filter all at once on input element)
- In this example: no communication between nodes of cluster (communication of one int at end to perform count() reduction)

```scala
val lines = spark.textFile("hdfs://cs149log.txt");
val lower = lines.map(_.toLowerCase);
val mobileViews = lower.filter(x => isMobileClient(x));
val howMany = mobileViews.count();
```
Wide dependencies

`groupByKey: RDD[(K,V)] → RDD[(K,Seq[V])]`

“Make a new RDD where each element is a sequence containing all values from the parent RDD with the same key.”

Wide dependencies = each partition of parent RDD referenced by multiple child RDD partitions

Challenges:
- Must compute all of RDD_A before computing RDD_B
  - Example: `groupByKey()` may induce all-to-all communication as shown above
- May trigger significant recomputation of ancestor lineage upon node failure
  (I will address resilience in a few slides)
Cost of operations depends on partitioning

\[
\text{join: } \text{RDD}[(K,V)], \text{RDD}[(K,W)] \rightarrow \text{RDD}[(K,(V,W))]
\]

Assume data in RDD_A and RDD_B are partitioned by key: hash username to partition id

RDD_A and RDD_B have different hash partitions: join creates wide dependencies

RDD_A and RDD_B have same hash partition: join only creates narrow dependencies
PartitionBy() transformation

- Inform Spark on how to partition an RDD
  - e.g., HashPartitioner, RangePartitioner

```scala
// create RDD from file system data
val lines = spark.textFile("hdfs://cs149log.txt");
val clientInfo = spark.textFile("hdfs://clientsupported.txt"); // (useragent, "yes"/"no")

// create RDD using filter() transformation on lines
val mobileViews = lines.filter(x => isMobileClient(x)).map(x => parseUserAgent(x));

// HashPartitioner maps keys to integers
val partitioner = spark.HashPartitioner(100);

// inform Spark of partition
// .persist() also instructs Spark to try to keep dataset in memory
val mobileViewPartitioned = mobileViews.partitionBy(partitioner).persist();
val clientInfoPartitioned = clientInfo.partitionBy(partitioner).persist();

// join useragents with whether they are supported or not supported
// Note: this join only creates narrow dependencies due to the explicit partitioning above
val joined = mobileViewPartitioned.join(clientInfoPartitioned);
```

- .persist():
  - Inform Spark this RDD’s contents should be retained in memory
  - .persist(RELIABLE) = store contents in durable storage (like a checkpoint)
PageRank Performance

Hadoop: 171 seconds
Basic Spark: 72 seconds
Spark + Controlled Partitioning: 23 seconds
Scheduling Spark computations

Actions (e.g., save()) trigger evaluation of Spark lineage graph.

Stage 1 Computation: do nothing since input already materialized in memory
Stage 2 Computation: evaluate map in fused manner, only actually materialize RDD F
Stage 3 Computation: execute join (could stream the operation to disk, do not need to materialize )
Implementing resilience via lineage

- **RDD transformations are bulk, deterministic, and functional**
  - Implication: runtime can always reconstruct contents of RDD from its lineage (the sequence of transformations used to create it)
  - Lineage is a log of transformations
  - Efficient: since the log records bulk data-parallel operations, overhead of logging is low (compared to logging fine-grained operations, like in a database)

```scala
// create RDD from file system data
val lines = spark.textFile("hdfs://cs149log.txt");

// create RDD using filter() transformation on lines
val mobileViews = lines.filter((x: String) => isMobileClient(x));

// 1. create new RDD by filtering only Chrome views
// 2. for each element, split string and take timestamp of page view (first element)
// 3. convert RDD To a scalar sequence (collect() action)
val timestamps = mobileView.filter(_.contains("Chrome"))
  .map(_.split(" ")(0));
```

Stanford CS149, Winter 2019
Upon node failure: recompute lost RDD partitions from lineage

val lines = spark.textFile("hdfs://cs149log.txt");
val mobileViews = lines.filter((x: String) => isMobileClient(x));
val timestamps = mobileView.filter(_.contains("Chrome"))
        .map(_.split(" ")(0));

Must reload required subset of data from disk and recompute entire sequence of operations given by lineage to regenerate partitions 2 and 3 of RDD timestamps.

Note: (not shown): file system data is replicated so assume blocks 2 and 3 remain accessible to all nodes
Upon node failure: recomputing lost RDD partitions from lineage

```scala
var lines = spark.textFile("hdfs://cs149log.txt");
var mobileViews = lines.filter((x: String) => isMobileClient(x));
var timestamps = mobileViews.filter(_.contains("Chrome")).map(_.split(" ")(0));
```

Must reload required subset of data from disk and recompute entire sequence of operations given by lineage to regenerate partitions 2 and 3 of RDD timestamps.

Note: (not shown): file system data is replicated so assume blocks 2 and 3 remain accessible to all nodes upon node failure: recompute lost RDD partitions from lineage
Spark performance

HadoopBM = Hadoop Binary In-Memory (convert text input to binary, store in in-memory version of HDFS)

Q. Wait, the baseline parses text input in each iteration of an iterative algorithm? A. Yes.

Anything else puzzling here?

HadoopBM’s first iteration is slow because it runs an extra Hadoop job to copy binary form of input data to in memory HDFS

Accessing data from HDFS, even if in memory, has high overhead:
- Multiple mem copies in file system + a checksum
- Conversion from serialized form to Java object

(100GB of data on a 100 node cluster)
Caution: "scale out" is not the entire story

- Distributed systems designed for cloud execution address many difficult challenges, and have been instrumental in the explosion of "big-data" computing and large-scale analytics
  - Scale-out parallelism to many machines
  - Resiliency in the face of failures
  - Complexity of managing clusters of machines
- But scale out is not the whole story:

20 Iterations of Page Rank

<table>
<thead>
<tr>
<th>scalable system</th>
<th>cores</th>
<th>twitter</th>
<th>uk-2007-05</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphChi [10]</td>
<td>2</td>
<td>3160s</td>
<td>6972s</td>
</tr>
<tr>
<td>Stratosphere [6]</td>
<td>16</td>
<td>2250s</td>
<td>-</td>
</tr>
<tr>
<td>X-Stream [17]</td>
<td>16</td>
<td>1488s</td>
<td>-</td>
</tr>
<tr>
<td>Spark [8]</td>
<td>128</td>
<td>857s</td>
<td>1759s</td>
</tr>
<tr>
<td>Giraph [8]</td>
<td>128</td>
<td>596s</td>
<td>1235s</td>
</tr>
<tr>
<td>GraphLab [8]</td>
<td>128</td>
<td>249s</td>
<td>833s</td>
</tr>
<tr>
<td>GraphX [8]</td>
<td>128</td>
<td>419s</td>
<td>462s</td>
</tr>
<tr>
<td>Single thread (SSD)</td>
<td>1</td>
<td>300s</td>
<td>651s</td>
</tr>
<tr>
<td>Single thread (RAM)</td>
<td>1</td>
<td>275s</td>
<td>-</td>
</tr>
</tbody>
</table>

Further optimization of the baseline brought time down to 110s

["Scalability! At what COST?" McSherry et al. HotOS 2015]
Caution: “scale out” is not the entire story

Label Propagation
[McSherry et al. HotOS 2015]

<table>
<thead>
<tr>
<th>Scalable System</th>
<th>Cores</th>
<th>Twitter</th>
<th>Uk-2007-05</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stratosphere [6]</td>
<td>16</td>
<td>950s</td>
<td>-</td>
</tr>
<tr>
<td>X-Stream [17]</td>
<td>16</td>
<td>1159s</td>
<td>-</td>
</tr>
<tr>
<td>Spark [8]</td>
<td>128</td>
<td>1784s</td>
<td>-</td>
</tr>
<tr>
<td>Giraph [8]</td>
<td>128</td>
<td>200s</td>
<td>∨ 8000s</td>
</tr>
<tr>
<td>GraphLab [8]</td>
<td>128</td>
<td>242s</td>
<td>714s</td>
</tr>
<tr>
<td>GraphX [8]</td>
<td>128</td>
<td>251s</td>
<td>800s</td>
</tr>
</tbody>
</table>

Single thread (SSD) | 1 | 153s | 417s |

BID Data Suite (1 GPU accelerated node)
[Canny and Zhao, KDD 13]

Page Rank

BID Data Suite (1 GPU accelerated node)
[Canny and Zhao, KDD 13]

Latency Dirichlet Allocation (LDA)

from McSherry 2015:

“The published work on big data systems has fetishized scalability as the most important feature of a distributed data processing platform. While nearly all such publications detail their system’s impressive scalability, few directly evaluate their absolute performance against reasonable benchmarks. To what degree are these systems truly improving performance, as opposed to parallelizing overheads that they themselves introduce?”

COST = “Configuration that Outperforms a Single Thread”

Perhaps surprisingly, many published systems have unbounded COST—i.e., no configuration outperforms the best single-threaded implementation—for all of the problems to which they have been applied.
Performance improvements to Spark

- With increasing DRAM sizes and faster persistent storage (SSD), there is interest in improving the CPU utilization of Spark applications
  - Goal: reduce “COST”

- Efforts looking at adding efficient code generation to Spark ecosystem (e.g., generate SIMD kernels, target accelerators like GPUs, etc.) to close the gap on single node performance
  - RDD storage layouts must change to enable high-performance SIMD processing (e.g., struct of arrays instead of array of structs)
  - See Spark’s Project Tungsten, Weld [Palkar Cidr ’17], IBM’s SparkGPU

- High-performance computing ideas are influencing design of future performance-oriented distributed systems
  - Conversely: the scientific computing community has a lot to learn from the distributed computing community about elasticity and utility computing
Spark summary

- Introduces opaque sequence abstraction (RDD) to encapsulate intermediates of cluster computations (previously... frameworks like Hadoop/MapReduce stored intermediates in the file system)
  - Observation: “files are a poor abstraction for intermediate variables in large-scale data-parallel programs”
  - RDDs are read-only, and created by deterministic data-parallel operators
  - Lineage tracked and used for locality-aware scheduling and fault-tolerance (allows recomputation of partitions of RDD on failure, rather than restore from checkpoint *)

  - Bulk operations allow overhead of lineage tracking (logging) to be low.

- Simple, versatile abstraction upon which many domain-specific distributed computing frameworks are being implemented.
  - See Apache Spark project: spark.apache.org

* Note that .persist(RELIABLE) allows programmer to request checkpointing in long lineage situations.
Modern Spark ecosystem

Compelling feature: enables integration/composition of multiple domain-specific frameworks
(since all collections implemented under the hood with RDDs and scheduled using Spark scheduler)

- Spark SQL
  ```scala
  sqlCtx = new HiveContext(sc)
  results = sqlCtx.sql("SELECT * FROM people")
  names = results.map(lambda p: p.name)
  ```

  Interleave computation and database query
  Can apply transformations to RDDs produced by SQL queries

- Spark MLlib
  ```scala
  points = spark.textFile("hdfs://...").map(parsePoint)
  model = KMeans.train(points, k=10)
  ```

  Machine learning library build on top of Spark abstractions.

- Spark GraphX
  ```scala
  graph = Graph(vertices, edges)
  messages = spark.textFile("hdfs://...")
  graph2 = graph.joinVertices(messages) {
    (id, vertex, msg) => ...
  }
  ```

  GraphLab-like library built on top of Spark abstractions.