Lecture 22:

Spark

(leveraging bulk-granularity program structure)
Yeah Yeah Yeahs
Sacrilege
(Mosquito)

“In-memory performance and fault-tolerance across a cluster. No way!”
- Karen O
Review: 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);
```

Two loads, one store per math op (arithmetic intensity = 1/3)

Two loads, one store per math op (arithmetic intensity = 1/3)

Overall arithmetic intensity = 1/3

Four loads, one store per 3 math ops (arithmetic intensity = 3/5)

The transformation of the code in program 1 to the code in program 2 is called “loop fusion”
Review: why did we perform this transform?

Program 1

```c
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];
float weights[] = {1.0/3, 1.0/3, 1.0/3};

// blur image horizontally
for (int j=0; j<HEIGHT; j++) {
    float tmp = 0.f;
    for (int i=0; i<WIDTH; i++)
        tmp += input[j*(WIDTH+2) + i] * weights[i];
    tmp_buf[j*WIDTH + i] = tmp;
}

// blur tmp_buf vertically
for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++)
        output[j*WIDTH + i] = tmp;
}
```

Program 2

```c
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (CHUNK_SIZE+2)];
float output[WIDTH * HEIGHT];
float weights[] = {1.0/3, 1.0/3, 1.0/3};

for (int j=0; j<HEIGHT; j++) {
    float tmp = 0.f;
    for (int i=0; i<WIDTH; i++)
        tmp += tmp_buf[(j+i)*WIDTH + i] * weights[i];
    output[(j+i)*WIDTH + i] = tmp;
}
```
Both of the previous examples involved globally restructuring the order of computation to improve produce-consumer locality.

(improve arithmetic intensity of program)
A log of page views on the 418 website.
Let’s say 418 gets very popular…

We’re taking the 1989 Tour to the Parallelism Computation!

You kids might want to learn to handle at least two directions before you take a stab at 16 cores. Count me in.

Your name might be Swift, but your code can’t keep up with ours.
The log of page views gets quite large...

Assume 15418log.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 15418log.txt are distributed evenly in blocks across the cluster
The respective agents of the new Parallelism Competition entrants want to know a bit more about the fans tracking the competition on the 418 site…

For example: “What type of mobile phone are all my client’s fans using?”
A simple programming model

```c
t// called once per line in input file by runtime
t// input: string (line of input file)
t// output: adds (user_agent, 1) entry to list
tvoid 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);
}

t// called once per unique key (user_agent) in results
	// values is a list of values associated with the given key

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

// iterator over lines of text file
	LineByLineReader input("hdfs://15418log.txt");

// stores output
	Writer output("hdfs://...");

// do stuff
	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

```cpp
// 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;
}
```

```cpp
LineByLineReader input("hdfs://15418log.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

<table>
<thead>
<tr>
<th>Block</th>
<th>Block</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Idea 2: data distribution based assignment: Each node processes lines in blocks of input file that are stored locally.
Steps 2 and 3: gathering data, running the reducer

// called once per line in file
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);
}

// 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://15418log.txt");
Writer output("hdfs://...");
runMapReduceJob(mapper, reducer, input, output);

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

Node 0

Node 1

Node 2

Node 3
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?

Example:
Assign Safari iOS to Node 0

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

Example:
Assign Safari iOS to Node 0
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
  - Generalize structure to DAGs (see DryadLINQ)

- **Iterative algorithms must load from disk each iteration**
  - Recall last lecture on graph processing:

```java
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]);
}
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)
Resilient distributed dataset (RDD)

Spark’s key programming abstraction:
- Read-only 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

```
// create RDD from file system data
var lines = spark.textFile("hdfs://15418log.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://15418log.txt")
  .filter(x => isMobileClient(x))
  .map(x => (parseUserAgent(x),1))
  .reduceByKey((x,y) => x+y)
  .collect();

```
// Lineage:
// Sequence of RDD operations needed to compute output
```
Another Spark program

// create RDD from file system data
var lines = spark.textFile("hdfs://15418log.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();
3.2.2 PageRank

The algorithm iteratively updates a rank for each document by adding up contributions from documents that link to it. On each iteration, each document receives contributions from its neighbors, where a contribution is the fraction of the rank that it sends to its neighbors, where a neighbor is a document that links to it.

The PageRank algorithm can be implemented in Scala using Spark's RDD (Resilient Distributed Dataset) API. Here is a simplified example of how to implement the PageRank algorithm using Spark:

```scala
// Define the number of iterations
val ITERATIONS = 10

// Load the web graph as an RDD of (URL, outlinks) pairs
val links = spark.textFile(...).map(...).

// Initialize ranks for each URL to be 1/N, where N is the total number of URLs
val ranks = links.map(url => (url, 1.0 / N)).persist()

for (i <- 1 to ITERATIONS) {
  // Sum contributions by URL to get new ranks
  val contribs = links.join(ranks).flatMap {
    case (url, (links, rank)) =>
      // Each URL sends a contribution of rank / outlinks to each of its neighbors
      // The contribution to each neighbor is the fraction of the rank that it receives
      links.mapToURL(url).map { neighborLink =>
        (neighborLink, rank / links.count)}
  }.

  // Update ranks
  ranks = contribs.reduceByKey((x, y) => x + y)
}
```

The PageRank algorithm converges after a few iterations, and the rank of each URL can be used to rank documents for search results. The PageRank algorithm also allows for speedup, as we show in Section 6.1.
How do we implement RDDs?
In particular, how should they be stored?

var lines = spark.textFile("hdfs://15418log.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?

var lines = spark.textFile("hdfs://15418log.txt");
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();

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

```
var lines = spark.textFile("hdfs://15418log.txt");
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```

Black lines show dependencies between RDD partitions.
Implementing sequence of RDD ops efficiently

```scala
var lines = spark.textFile("hdfs://15418log.txt");
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```

Recall “loop fusion” from start 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.toLowerCase();
    if (isMobileClient(lower))
        count++;
}
```
A simple interface for RDDs

```scala
var lines = spark.textFile("hdfs://15418log.txt");
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var howMany = mobileViews.count();
```

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

RDD::hasMoreElements() {
    parent.hasMoreElements();
}

// overloaded since no parent exists
RDDFromTextFile::hasMoreElements() {
    return !inputFile.eof();
}

RDD::next() {
    return inputFile.readLine();
}

RDDFromMap::next() {
    var el = parent.next();
    return el.toLowerCase();
}

RDDFromFilter::next() {
    while (parent.hasMoreElements()) {
        var el = parent.next();
        if (isMobileClient(el))
            return el;
    }
}
```
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)

```javascript
var lines = spark.textFile("hdfs://15418log.txt");
var lower = lines.map(_.toLower());
var mobileViews = lower.filter(x => isMobileClient(x));
var 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 recompilation of ancestor lineage upon node failure (will address resilience in a few slides)
Cost of operations depends on partitioning

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

Assume data in \( \text{RDD\_A} \) and \( \text{RDD\_B} \) 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

Assume data in \( \text{RDD\_A} \) and \( \text{RDD\_B} \) are partitioned by key: hash username to partition id
PartitionBy() transformation

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

```java
// create RDD from file system data
var lines = spark.textFile("hdfs://15418log.txt");
var clientInfo = spark.textFile("hdfs://clientssupported.txt"); // (useragent, "yes"/"no")

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

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

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

// join agents with whether they are supported or not supported
// Note: this join only creates narrow dependencies
void joined = mobileViewPartitioned.join(clientInfoPartitioned);
```

- .persist():
  - Inform Spark this RDD materialized contents should be retained in memory
  - .persist(RELIABLE) = store contents in durable storage (like a checkpoint)
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 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
var lines = spark.textFile("hdfs://15418log.txt");

// create RDD using filter() transformation on lines
var 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)
var timestamps = mobileView.filter(_.contains("Chrome")).map(_.split(" ")(0));
```
Upon node failure: recomputed lost RDD partitions from lineage

```scala
var lines = spark.textFile("hdfs://15418log.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: recomputing lost RDD partitions

```scala
var lines = spark.textFile("hdfs://15418log.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.
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
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>

Table 2 compares the reported times from several systems against a single-threaded implementations from SSD and from RAM.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes</td>
<td>41,652,230</td>
<td>105,896,555</td>
</tr>
<tr>
<td>edges</td>
<td>1,468,365,182</td>
<td>3,738,733,648</td>
</tr>
<tr>
<td>size</td>
<td>5.76GB</td>
<td>14.72GB</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>$\geq 8000$s</td>
</tr>
<tr>
<td>Giraph [8]</td>
<td>128</td>
<td>200s</td>
<td>$\geq 8000$s</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>
<tr>
<td>Single thread (SSD)</td>
<td>1</td>
<td>153s</td>
<td>417s</td>
</tr>
</tbody>
</table>

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 efficient code generation to Spark ecosystem (e.g., generate SIMD kernels, target accelerators like GPUs, etc.) to close the gap on single node performance
  - See Spark’s Project Tungsten

- 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)

```
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

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

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

Spark SQL

Spark MLlib

Spark GraphX

Machine learning library build on top of Spark abstractions.

GraphLab-like library built on top of Spark abstractions.
If you enjoyed today’s topic…

- I recommend looking at Legion
  - legion.stanford.edu
- Designed from Supercomputing perspective
  - Spark was designed from a distributed computing perspective
- Key idea: programming via collections called “logical regions”
  - Systems provides operators for hierarchically partitioning contents of regions
  - Tasks operate on regions with certain privileges (read/write/etc.)
  - Scheduler schedules tasks based on privileges to avoid race conditions
- Another of example of bulk-granularity programming
  - Overheads are amortized over very large data-parallel operations