Programming at Scale: Dataflow

cs378
Today

Questions?

Administrivia
• Project Proposal Due Soon!

Agenda:
• Dataflow Wrap up
• Start talking about Consistency at scale
Spark faux quiz (5 min, any 2):

- What is the difference between transformations and actions in Spark?
- Spark supports a persist API. When should a programmer want to use it? When should she [not] use use the “RELIABLE” flag?
- List aspects of Spark’s design that help/hinder multi-core parallelism relative to MapReduce. If the issue is orthogonal, explain why.
- Compare and contrast fault tolerance guarantees of Spark to those of MapReduce. How are[n’t] the mechanisms different?
- Compare/contrast the abstractions for parallelism in Spark/MapReduce
- For what kinds of workloads will Spark/MR have different/similar performance?
- Why does Spark expose control over caching RDDs in memory to the programmer?
- What’s a “wide” dependence? A “narrow” one? How do these ideas relate to fault tolerance in Spark?
- Is Spark a good system for indexing the web? For computing page rank over a web index? Why [not]?
Review: Scale: Goal
Infrastructure is hard to get right

1. How do we distribute the searchable files on our machines?
2. What if our webserver goes down?
3. What if a StringFinder machine dies? How would you know it was dead?
4. What if marketing comes and says, “well, we also want to show pictures of the earth from space too! Ooh..and the moon too!”
Dataflow Engines

Programming model + infrastructure
Write programs that run on lots of machines
Automatic parallelization and distribution
Fault-tolerance
I/O and jobs Scheduling
Status and monitoring

Key Ideas:
All modern “big data” platforms are dataflow engines!

Differences:
1. what graph structures are allowed?
2. How does this impact programming model?
Spark (2012) Background

Commodity clusters: important platform

- **In industry:** search, machine translation, ad targeting, ...
- **In research:** bioinformatics, NLP, climate simulation, ...

Cluster-scale models (e.g. MR) de facto standard

- Fault tolerance through replicated durable storage
- Dataflow is the common theme

- Multi-core
- Iteration
Motivation

Programming models for clusters transform data flowing from stable storage to stable storage

E.g., MapReduce:

**Benefits of data flow:** runtime can decide where to run tasks and can automatically recover from failures
Iterative Computations: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each page’s rank to
   \[ \Sigma_{i \in \text{neighbors}} \frac{\text{rank}_i}{|\text{neighbors}_i|} \]

```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(_ + _)
}
```
Iterative Computations: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each page’s rank to
   \[ \Sigma_{i \in \text{neighbors}} \frac{\text{rank}_i}{|\text{neighbors}_i|} \]

```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(_ + _)
}
```

Solution: augment data flow model with “resilient distributed datasets” (RDDs)
Programming Model

• Resilient distributed datasets (RDDs)
  • Immutable collections partitioned across cluster that can be rebuilt if a partition is lost
  • Created by transforming data in stable storage using data flow operators (map, filter, group-by, ...)
  • Can be cached across parallel operations

• Parallel operations on RDDs
  • Reduce, collect, count, save, ...

• Restricted shared variables
  • Accumulators, broadcast variables
Example: Log Mining

• Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()
cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
... 
```

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

- RDDs maintain *lineage* information that can be used to reconstruct lost partitions

- Ex:
  
  ```scala
  cachedMsgs = textFile(...).filter(_.contains("error")).map(_.split('t')(2)).persist()
  ```
```
lines = LOAD '/user/hadoop/HDFS_File.txt' AS (line:chararray);
words = FOREACH lines GENERATE FLATTEN(TOKENIZE(line)) as word;
grouped = GROUP words BY word;
wordcount = FOREACH grouped GENERATE group, COUNT(words);
DUMP wordcount;
```

---

### Systems

**Data-Parallel Computation Systems**
- Hadoop
- HDFS
- S3
- Pig, Hive
- Spark
- SQL Server
- Cosmos
- Azure
- SQL
- Dryad
- DryadLINQ
- HPC
- Sawzall
- LINQ, SQL

---

### Execution

**Parallel Databases**
- SQL
- Sawzall
- ≈SQL
- Pig, Hive
- DryadLINQ

---

### Language

**Parallel Databases**
- SQL
- ≈SQL
- LINQ, SQL

---

### Storage

**Parallel Databases**
- BigTable
- S3
- SQL Server

---

```
count: table sum of int;
total: table sum of float;
sum_of_squares: table sum of float;
x: float = input;
emit count <- 1;
emit total <- x;
emit sum_of_squares <- x * x;
```
Background: Collections and Iterators

class Collection<T> : IEnumerable<T>;

public interface IEnumerable<T>  {
    IEnumerator<T> GetEnumerator();
}

public interface IEnumerator<T> {
    T Current { get; }
    bool MoveNext();
    void Reset();
}
DryadLINQ = LINQ + Dryad

```csharp
Collection<T> collection;
bool IsLegal(Key k);
string Hash(Key);

var results = from c in collection
    where IsLegal(c.key)
    select new { Hash(c.key), c.value};
```
Programming Model

Where
Select
GroupBy
OrderBy
Aggregate
Join
Apply
Materialize
public static IQueryable<Pair> Histogram(
    IQueryable<LineRecord> input, int k)
{
    var words = input.SelectMany(x => x.line.Split(' '));
    var groups = words.GroupBy(x => x);
    var counts = groups.Select(x => new Pair(x.Key, x.Count()));
    var ordered = counts.OrderByDescending(x => x.count);
    var top = ordered.Take(k);
    return top;
}
RDDS

• Immutable, partitioned, logical collection of records
  • Need not be materialized
  • contains information to rebuild a dataset
    • Partitioning can be based on a key
  • Built using bulk transformations on other RDDs
  • Can be cached for future reuse

Transformations
(define a new RDD)
map
filter
sample
union
groupByKey
reduceByKey
join
persist/cache
...

Parallel operations
(return a result to driver)
reduce
collect
count
save
lookupKey
...

Where
Select
GroupBy
OrderBy
Aggregate
Join
Apply
Materialize
## RDDs vs Distributed Shared Memory

<table>
<thead>
<tr>
<th>Concern</th>
<th>RDDs</th>
<th>Distr. Shared Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
<td>Fine-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Writes</td>
<td>Bulk transformations</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Consistency</td>
<td>Trivial (immutable)</td>
<td>Up to app / runtime</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Fine-grained and low-overhead using lineage</td>
<td>Requires checkpoints and program rollback</td>
</tr>
<tr>
<td>Straggler mitigation</td>
<td>Possible using speculative execution</td>
<td>Difficult</td>
</tr>
<tr>
<td>Work placement</td>
<td>Automatic based on data locality</td>
<td>Up to app (but runtime aims for transparency)</td>
</tr>
</tbody>
</table>
Summary

Dataflow key enabler for cluster-scale parallelism

Key issues become runtime’s responsibility
  - Data movement
  - Scheduling
  - Fault-tolerance
MapReduce is sub-optimal

Modern DBMSs: hash + B-tree indexes to accelerate data access.
   Indexes are user-defined
   Could MR do this?

No query optimizer! (oh my, terrible...but good for researchers! 😊)

Skew: wide variance in distribution of keys
   E.g. “the” more common than “zyzzyva”

Materializing splits
   N=1000 mappers → M=500 keys = 500,000 local files
   500 reducer instances “pull” these files
   DBMSs push splits to sockets (no local temp files)
MapReduce: novel && feature-poor

- Partitioning data sets (map) == Hash join
- Parallel aggregation == reduce
- User-supplied functions differentiates from SQL:
  - POSTGRES user functions, user aggregates
  - PL/SQL: Stored procedures
  - Object databases

Absent features:
- Indexing
- Update operator
- Transactions
- Integrity constraints, referential integrity
- Views
Why is MapReduce backwards?

Map == group-by
Reduce == aggregate

```
SELECT job, COUNT(*) as "numemps"
FROM employees
WHERE salary > 1000
GROUP BY job;
```

• Where is the aggregate in this example?
• Is the DBMS analogy clear?
Why is MapReduce backwards?

Schemas are good (what’s a schema?)
Separation of schema from app is good (why?)
High-level access languages are good (why?)