Announcements

• Final Exam in class Friday
  – 60min exam – focus will be on second half of class
  – Wednesday – review what’s on the exam
  – Thursday section – more review

• HW Grades through HW6 are published
  – HW6 is out of 100 points but will be scaled to 21
Announcements

• Homework breakdown (50%) of grade
  – HW1 (sqlite) 20 pt
  – HW2 (sqlite) 20 pt
  – HW3 (sql sever) 27 pt
  – H4 (RA,RC,Datalog) 15pt
  – H6 (E/R BCNF) 21pt
  – HW7 30 pt
Lifecycle of a MR Program

1. Read a lot of data and parse into (key, value) pairs
2. Map: extract something you care about from each (key, value) pair
3. Shuffle output from mappers
   - done internally by implementation
4. Reduce: aggregate, summarize, filter, transform
5. Write the results to files

Paradigm stays the same, change map and reduce functions for different problems
Issues with MapReduce

• Difficult to write more complex queries

• Need multiple MapReduce jobs: dramatically slows down because it writes all results to disk
Over view of MapReduce

![Diagram of MapReduce process]

- Input Files in HDFS
  - Mapper
  - Partitioner
  - Combiner
  - Mapper
  - Partitioner
  - Combiner
  - Mapper
  - Partitioner
  - Combiner
  - Mapper
  - Partitioner
  - Combiner
- Shuffle Phase
- Sorting happens in the Reducer node before the Reducer consumes its input key/value instances
- Output Files in HDFS
  - <K1,V1>
  - <K2,V2>
  - <K2,V2>
  - <K2,V2>
  - <K3,V3>
Spark

• Open source system from Berkeley
• Distributed processing over HDFS
• Differences from MapReduce:
  – Multiple steps, including iterations
  – Stores intermediate results in main memory
  – Supports SQL
• Details: http://spark.apache.org/examples.html
• Free Trail: https://community.cloud.databricks.com/
Spark Interface

• Spark supports a Scala interface
  • Scala = ext of Java with functions/closures
    – will show Scala/Spark examples shortly…

• Spark also supports a SQL interface
  • It compiles SQL into Scala
    – Best of both world: programmatic and SQL APIs
RDD

- RDD = Resilient Distributed Datasets
  - A distributed relation, together with its *lineage*
  - Lineage = expression that says how that relation was computed = a relational algebra plan
- Spark stores intermediate results as RDD
- If a server crashes, its RDD in main memory is lost. However, the driver (=master node) knows the lineage, and will simply recompute the lost partition of the RDD
Programming in Spark

• A Spark/Scala program consists of:
  – Transformations (map, reduce, join…). Lazy
  – Actions (count, reduce, save…). Eager

• RDD[T] = an RDD collection of type T
  – Partitioned, recoverable (through lineage), not nested

• Seq[T] = a Scala sequence
  – Local to a server, may be nested
Example

Given a large log file hdfs://logfile.log retrieve all lines that:

• Start with “ERROR”
• Contain the string “sqlite”

```scala
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()
```
Example

Given a large log file hdfs://logfile.log retrieve all lines that:
• Start with “ERROR”
• Contain the string “sqlite”

```scala
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()
```
MapReduce Again…

Steps in Spark resemble MapReduce:

- `col.filter(p)` applies in parallel the predicate `p` to all elements `x` of the partitioned collection, and returns those `x` where `p(x) = true`

- `col.map(f)` applies in parallel the function `f` to all elements `x` of the partitioned collection, and returns a new partitioned collection
Scala Primer

• Functions with one argument:
  `_ .contains("sqlite")
  _ > 6

• Functions with more arguments
  (x => x.contains("sqlite"))
  (x => x > 6)
  ((x,y) => x+3*y)

• Closures (functions with variable references):
  var x = 5;   rdd .filter(_ > x)
  var s = "sqlite";  rdd .filter(x => x.contains(s))
Persistenace

```scala
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()
```

If any server fails before the end, then Spark must restart
Persisten

tlines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()

If any server fails before the end, then Spark must restart
Persistence

If any server fails before the end, then Spark must restart

```scala
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()
```

Spark can recompute the result from errors

```scala
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
errors.persist()
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()
```
Persistence

If any server fails before the end, then Spark must restart.

```scala
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
sqlerrors.collect()
```

Spark can recompute the result from errors.
Example

R = spark.textFile("R.csv").map(parseRecord).persist()
S = spark.textFile("S.csv").map(parseRecord).persist()
RB = R.filter((a,b) => b > 200).persist()
SC = S.filter((a,c) => c < 100).persist()
J = RB.join(SC).persist()
J.count();
Programming in Spark

• A Spark/Scala program consists of:
  – Transformations (map, reduce, join…). Lazy
  – Actions (count, reduce, save…). Eager

• RDD[T] = an RDD collection of type T
  – Partitioned, recoverable (through lineage), not nested

• Seq[T] = a Scala sequence
  – Local to a server, may be nested
### Transformations:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(f : T =&gt; U)</code></td>
<td>RDD[T] =&gt; RDD[U]</td>
</tr>
<tr>
<td><code>flatMap(f: T =&gt; Seq[U])</code></td>
<td>RDD[T] =&gt; RDD[U]</td>
</tr>
<tr>
<td><code>filter(f:T=&gt;Bool)</code></td>
<td>RDD[T] =&gt; RDD[T]</td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td>RDD[(K,V)] =&gt; RDD[(K,Seq[V])]</td>
</tr>
<tr>
<td><code>reduceByKey(F:(V,V) =&gt; V)</code></td>
<td>RDD[(K,V)] =&gt; RDD[(K,V)]</td>
</tr>
<tr>
<td><code>union()</code></td>
<td>(RDD[T],RDD[T]) =&gt; RDD[T]</td>
</tr>
<tr>
<td><code>join()</code></td>
<td>(RDD[(K,V)],RDD[(K,W)]) =&gt; RDD[(K,(V,W))]</td>
</tr>
<tr>
<td><code>cogroup()</code></td>
<td>(RDD[(K,V)],RDD[(K,W)]) =&gt; RDD[(K,(Seq[V],Seq[W]))]</td>
</tr>
<tr>
<td><code>crossProduct()</code></td>
<td>(RDD[T],RDD[U]) =&gt; RDD[(T,U)]</td>
</tr>
</tbody>
</table>

### Actions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td>RDD[T] =&gt; Long</td>
</tr>
<tr>
<td><code>collect()</code></td>
<td>RDD[T] =&gt; Seq[T]</td>
</tr>
<tr>
<td><code>reduce(f:(T,T)=&gt;T)</code></td>
<td>RDD[T] =&gt; T</td>
</tr>
<tr>
<td><code>save(path:String)</code></td>
<td>Outputs RDD to a storage system e.g. HDFS</td>
</tr>
</tbody>
</table>
MapReduce ~> Spark

- input into an RDD
- map phase becomes .flatMap
- shuffle & sort becomes .groupByKey
- reduce becomes another .flatMap
- save output to HDFS
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = textFile
    .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b);
counts.saveAsTextFile("hdfs://...");

val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...");
SQL $\Rightarrow$ Spark

- You know enough to execute SQL on Spark!
- Idea: (1) SQL to RA + (2) RA on Spark
  - $\sigma =$ filter
  - $\pi =$ map
  - $\gamma =$ groupByKey
  - $\times =$ crossProduct
  - $\Join =$ join
- Spark SQL does small optimizations to RA
- Also chooses btw broadcast and parallel joins
Spark APIs: SQL

// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
Google Dataflow

- Similar to Spark/Scala
- Allows you to lazily build pipelines and then execute them
- Much simpler than multi-job MapReduce
Dataflow Example Pipeline

Counting words again…

CSE 344 - Summer 2017
Dataflow Example Code

Pipeline p = Pipeline.create(options);

p.apply(TextIO.Read.from("gs://dataflow-samples/shakespeare/kinglear.txt"))

 .apply(ParDo.named("ExtractWords").of(new DoFn<String, String>() {
            @Override
            public void processElement(ProcessContext c) {
                for (String word : c.element().split("[^a-zA-Z']+")) {
                    if (!word.isEmpty()) {
                        c.output(word);
                    }
                }
            }
        }));
.apply(Count.<String>perElement())

.apply(MapElements.via(new SimpleFunction<KV<String, Long>, String>() {
    @Override
    public String apply(KV<String, Long> element) {
        return element.getKey() + ": " + element.getValue();
    }
}))

.apply(TextIO.Write.to("gs://my-bucket/counts.txt");

execute now
Where is Spark Used

[Diagram showing the integration of Spark with other systems like Couchbase, hdfs, s3, RDBMS, and Elasticsearch.]
Summary

- Parallel databases
  - Predefined relational operators
  - Optimization
  - Transactions
- MapReduce
  - User-defined map and reduce functions
  - Must implement/optimize manually relational ops
  - No updates/transactions
- Spark
  - Predefined relational operators
  - Must optimize manually
  - No updates/transactions
Summary cont.

- All of these technologies use **dataflow engines**:
  - Google Dataflow (on top of MapReduce)
  - Spark (on top of Hadoop)
  - AsterixDB (on top of Hyracks)
- Spark & AsterixDB map SQL to a dataflow pipeline
  - SQL -> RA -> dataflow operators (group, join, map)
  - could do the same thing for Google Dataflow
- None of these systems optimize RA very well (as of 2015)
  - Spark has no indexes
  - AsterixDB has indexes but no statistics
- Future work should improve that