Delta Processing for Social Analytics

James Thomas, Xinhao Yuan
Myeongjae Jeon, Jorgen Thelin,
Lidong Zhou
DSoAP System Architecture (from before)

- Retrieve tweets with plain text search and perform some analysis on them
Common Social Analytics Workflow

1. Plain text search to get target tweet set
2. Analysis within groups of tweets
   – e.g. each group has all tweets with a certain pair of words co-occurring
3. Finally global analysis (e.g. sort the groups by some field)
   – e.g. sort co-occurrence pairs by # of tweets
4. Refine search query if irrelevant results observed and repeat steps 1-3

Make this process faster, assuming fixed analysis
Motivation

- Honing in on target tweet set with plain text search

Search 1: “Java”
- Java programming language
- Java coffee
- Java Island

Search 2: “Java AND (NOT coffee) AND (NOT brew)”
- Java programming language
- Java Island

Search 3: “Java AND (Oracle OR JVM OR language ....)”
- Java programming language
Problem

• Only realize tweet set is bad at end of computation pipeline

<table>
<thead>
<tr>
<th>Co-occurrence pair</th>
<th># of appearances</th>
</tr>
</thead>
<tbody>
<tr>
<td>(JVM, JIT)</td>
<td>100</td>
</tr>
<tr>
<td>(Philippines, island)</td>
<td>55</td>
</tr>
<tr>
<td>(Scala, functional)</td>
<td>30</td>
</tr>
</tbody>
</table>

• Must rerun whole computation pipeline on updated tweet set, even though many tweets shared
Solution

- Keep previous results, pass only deltas through computation pipeline and update previous results

### Source Deltas

<table>
<thead>
<tr>
<th>Operation</th>
<th>Tweet text</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete</td>
<td>“I love Java Island in the Philippines!”</td>
</tr>
<tr>
<td>Add</td>
<td>“The JIT in Oracle JVM 1.8.0.05 is tight!”</td>
</tr>
</tbody>
</table>

### Result Deltas

<table>
<thead>
<tr>
<th>Operation</th>
<th>Co-occurrence pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete</td>
<td>(Philippines, island)</td>
</tr>
<tr>
<td>Add</td>
<td>(JVM, JIT)</td>
</tr>
</tbody>
</table>

### Original Result Set

<table>
<thead>
<tr>
<th>Co-occurrence pair</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>(JVM, JIT)</td>
<td>100</td>
</tr>
<tr>
<td>(Philippines, island)</td>
<td>55</td>
</tr>
</tbody>
</table>

### New Result Set

<table>
<thead>
<tr>
<th>Co-occurrence pair</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>(JVM, JIT)</td>
<td>101</td>
</tr>
<tr>
<td>(Philippines, island)</td>
<td>54</td>
</tr>
</tbody>
</table>
Design Goals / Limits of Prior Work

1. Have good distributed computation characteristics, like fault/straggler tolerance, communication avoidance
   - Modify Spark without losing these properties

2. Efficient fine-grained updates to prior results
Design Goals Cont’d

3. Mix non-incrementalizable computations with incrementalizable in general-purpose computation framework

4. Easy to roll back to and apply deltas to previous result sets

![Diagram](image)
Spark Review

• Partitioned datasets (RDDs)
• One-to-one and shuffle dependencies (narrow and wide)
• DAG of dependencies
• Programmer-specified caching of datasets in memory

```python
rdd2 = 
  rdd1.filter([lambda])
  .groupByKey()
  .mapValues([lambda])
```

![Dependency DAG](image)

![Physical Execution](image)
Feeding Deltas through Computation

\[
\text{rdd2} = \text{rdd1}.\text{filter}(t=>t._1>5) \cdot \text{groupByKey()} \cdot \text{mapValues}(v=>v.\text{min})
\]

Case 1: one-to-many transforms of records only -- pass deltas through pipeline for output update

- (6, "james")
- (6, "james")
+ (1, "jamie")

Case 2: shuffle transforms (many-to-one) as well -- must save intermediate results

- (6, "james")
+ (1, "jamie")

Don’t know other keys in group 6, so can’t pass deltas through this stage without saving ShuffledRDD
Arbitrary map functions can be applied to groups, so we must pass entire old group as neg. delta and new group as pos. delta

```
rdd2 =
rdd1.filter(t=>t._1>5)
  .groupByKey()
  .mapValues(v=>v.min)
```
Target API (Design Goals 3 & 4)

• \(d = \text{new DeltaComputation}(\text{rdd})\)
  – \(\text{rdd}\) can be any map-filter-groupby computation

• \(\text{newD} = d.\text{applyDeltas}(\text{pos}, \text{neg})\)
  – \(\text{pos}, \text{neg}\) are RDDs that are deltas to the source dataset in the computation in \(d\)
  – \(d\) is not mutated, so we still have a handle to it and can apply other deltas

• \(\text{result} = \text{newD}.\text{getResult}()\)
  – \(\text{result}\) is an RDD on which any Spark transformations (incrementalizable or not) can be run
result = source.map(...).groupByKey().map(...)
  .groupByKey().filter()...

d = new DeltaComputation(result)
top10 = d.getResult().topK(10)
// top10 has bad results, refine query
(pos, neg) = [submit new text filter to storage layer, obtain +/- deltas to source]
dUpdated = d.applyDeltas(pos, neg)
newTop10 = dUpdated.getResult().topK(10)
// realize new query was completely wrong, go back to original query and refine it
(newPos, newNeg) = ...
dUpdated2 = d.applyDeltas(newPos, newNeg)
...
Generality of map-filter-groupby

• Our system can pass deltas through any map-filter-groupby computation
• All single-dataset computations can be expressed with these operations (c.f. MapReduce)
• Social analytics generally groups records and then performs map-like transformations on the groups, then groups again and repeats
Fine-grained Updates to RDDs

- Fast, space-efficient fine-grained updates required for deltas not easy in Spark due to dataset immutability
- IndexedRDD (AMP Lab) uses functional data structure idea to solve problem
- Space-efficiency means persisting old versions is cheap (design goal 4)

(fig. by Ankur Dave, AMP Lab)
Computation DAG Rewriter (Design Goals 1 & 2)

```scala
rdd2 = rdd1.filter([lambda])
  .groupByKey()
  .mapValues([lambda])

dc = new DeltaComputation(rdd2)
source = dc.getResult()

updated = dc.applyDeltas(pos, neg)
  .getResult()
```
Implementation Details

- Just a library – no modifications to Spark core
- ~300 LOC (DAG rewriter and modifications to IndexedRDD)
- Spark partitioners used to avoid shuffles for IndexedRDD state maintenance
Evaluation Methodology

• 1e7 records, uniformly distributed over G keys
• Computation:
  `records.map(...).groupByKey().map(...)`
• Feed D deltas into the computation
• Measure percentage improvement in runtime over redoing whole computation
Results

Increasing because fewer deltas passed through computation pipeline

<table>
<thead>
<tr>
<th>#groups</th>
<th>1e2</th>
<th>1e3</th>
<th>1e4</th>
<th>1e5</th>
<th>1e6</th>
<th>2.5e6</th>
<th>5e6</th>
<th>1e7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e2</td>
<td>87%</td>
<td>83%</td>
<td>79%</td>
<td>81%</td>
<td>71%</td>
<td>49%</td>
<td>42%</td>
<td>-48%</td>
</tr>
<tr>
<td>1e3</td>
<td>95%</td>
<td>91%</td>
<td>81%</td>
<td>85%</td>
<td>77%</td>
<td>47%</td>
<td>24%</td>
<td>-47%</td>
</tr>
<tr>
<td>1e4</td>
<td>94%</td>
<td>93%</td>
<td>90%</td>
<td>87%</td>
<td>70%</td>
<td>55%</td>
<td>15%</td>
<td>-2%</td>
</tr>
<tr>
<td>1e5</td>
<td>95%</td>
<td>94%</td>
<td>93%</td>
<td>89%</td>
<td>72%</td>
<td>49%</td>
<td>16%</td>
<td>7%</td>
</tr>
<tr>
<td>1e6</td>
<td>91%</td>
<td>92%</td>
<td>90%</td>
<td>86%</td>
<td>61%</td>
<td>42%</td>
<td>-2%</td>
<td>-4%</td>
</tr>
</tbody>
</table>

Good perf. even when most groups changed

Good perf. even at 25%-50% of dataset changed
General Applicability of Delta Engine

• Can be used on top of any datastore that can return deltas

• Can be used to improve performance of iterative computations where each iteration can work with deltas from previous iteration
Conclusion

• General purpose delta processing system built on Spark, with applications in social analytics

• Achieved design goals:
  – Retains good properties of Spark
  – Supports fine-grained updates to prior results
  – Supports mixing of incremental and non-incremental computation
  – Efficiently saves prior dataset versions for reuse
Future Work

• Support wider class of Spark operators (joins, etc.)

• Test performance when intermediate IndexedRDDs must be recomputed and when delta volume large for several stages
  – Decide intelligently in these cases whether full pipeline recomputation is faster

• Support updating the computation as well as the dataset
  – Identifying common subcomputations and running the deltas through them